SECURE AND TRUSTED EXECUTION FRAMEWORK

FOR VIRTUALIZED WORKLOADS

Srujan D. Kotikela

Dissertation Prepared for the Degree of

DOCTOR OF PHILOSOPHY

UNIVERSITY OF NORTH TEXAS

August 2018

APPROVED:

Krishna Kavi, Committee Chair
Mahadevan Gomathisankaran, Advisor
Song Fu, Committee Member
Hassan Takabi, Committee Member
Bill Buckles, Committee Member
Barrett Bryant, Chair of the Department of Computer Science and Engineering
Yan Huang, Interim Dean of the College of Engineering
Victor Prybutok, Dean of the Toulouse Graduate School

Kotikela, Srujan D. *Secure and Trusted Execution Framework for Virtualized Workloads*. Doctor of Philosophy (Computer Science and Engineering), August 2018, 124 pp., 1 table, 18 figures, 1 appendix, 163 numbered references.

In this dissertation, we have analyzed various security and trustworthy solutions for modern computing systems and proposed a framework that will provide holistic security and trust for the entire lifecycle of a virtualized workload. The framework consists of 3 novel techniques and a set of guidelines. These 3 techniques provide necessary elements for secure and trusted execution environment while the guidelines ensure that the virtualized workload remains in a secure and trusted state throughout its lifecycle. We have successfully implemented and demonstrated that the framework provides security and trust guarantees at the time of launch, any time during the execution, and during an update of the virtualized workload. Given the proliferation of virtualization from cloud servers to embedded systems, techniques presented in this dissertation can be implemented on most computing systems.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to the team that helped me become a qualified researcher: my advisor, my committee, my friends, and my family.

I am extremely grateful to my committee chair Dr. Krishna Kavi for working with me, providing necessary guidance for navigating the Ph.D. journey, and helping me determine my career goals. I sincerely thank him for countless hours of discussions, tireless reviewing, resourceful advising, and constant encouragement.

A special thanks to my advisor Dr. Mahadevan Gomathisankaran for inspiring me to pursue a Ph.D., patiently teaching me how to conduct research, kindly supporting me with necessary resources, and transforming me into a qualified researcher. I would like to thank the members of my Ph.D. committee Dr. Song Fu, Dr. Hassan Takabi, and Dr. Bill Buckles for their thoughtful participation, providing invaluable feedback on my research and dissertation.

I would also like to thank my co-authors and fellow students: Dr. Satyajeet Nimgaonkar, Patrick Kamongi, and Tawfiq Shah for always being there through the hard times and happy times alike, giving useful feedback, helping me sort through the haystack of crazy ideas, and encouraging me when I needed them the most.

Many thanks to Stephanie Deacon for her patience and diligently ensuring a smooth administrative process throughout my graduate studies. I would also like to thank my dear friends Sandeep Kedarasipalli, Nethravathi, Vishwanath and Isaac Chilton for their wise words, moral support, and for the occasional nudges that helped me stay focused and finish the task at hand.

Finally, I would like to express my heartfelt gratitude to my family: my parents and my sisters for always being there, supporting me with all they have, and helping me become the person I am today. Without their love, this dissertation would not have been accomplished.

Chapter 5 is supported in part by grant from NSF (#1128344) and the Net-centric Industry/University Cooperative Research Center (UNT IUCRC).

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS iii

LIST OF TABLES x

LIST OF FIGURES xi

CHAPTER 1 INTRODUCTION 1

1.1. Ubiquitous Computers 1

1.2. A Vision for Trusted and Secure Computing Systems 2

1.3. Methodology 3

1.3.1. Secure Execution 4

1.3.2. Trust and Integrity Measurement 5

1.3.3. Vulnerability Management 6

1.4. Summary 7

CHAPTER 2 BACKGROUND AND MOTIVATION 8

2.1. Hardware Security Architectures 8

2.2. Virtualization 10

2.2.1. Hardware Support for Virtualization 12

2.3. Trusted Computing 13

2.3.1. Extend Operation 14

2.3.2. Bind Operation 14

2.3.3. Seal Operation 15

2.3.4. Quote Operation 15

2.4. Cloud Computing 15

2.4.1. IaaS 16

2.4.2. PaaS 16

2.4.3. SaaS 17

2.5. Vulnerability Management 17

CHAPTER 3 VIRTUALIZATION BASED SECURE EXECUTION 19

3.1. Introduction 19

3.2. Proposed Framework 20

3.2.1. Xen Hypervisor/VMM 20

3.2.2. MoCo VM 21

3.2.3. Application VM 22

3.2.4. Event Trigger Mechanism 22

3.2.5. Secure Architecture Plug-in Interface 22

3.2.6. Monitor 23

3.2.7. Controller 23

3.2.8. Secure Application 24

3.2.9. Attacker VM 24

3.2.10. Framework Functioning 24

3.2.11. Attack Model 25

3.3. Implementation 26

3.3.1. Source Code Implementation 28

3.4. CTrust Framework 32

3.4.1. CTrust Architecture 32

3.4.2. Security Analysis 34

3.5. Related Work 34

3.5.1. Hardware based secure execution 34

3.5.2. Process-level Isolation using Virtualization 34

3.5.3. Quebes OS 35

3.5.4. Singularity 35

3.5.5. Framework for Design Validation of Security Architectures 36

3.6. Future Work 36

3.7. Summary 37

CHAPTER 4 RACE-FREE ON-DEMAND INTEGRITY MEASUREMENT 38
4.1. Introduction 38
4.2. Related Work 40
4.2.1. SRTM and DRTM 40
4.2.2. AMD SVM and Intel TXT 42
4.2.3. Measured Boot 43
4.2.4. Trusted Hypervisor 44
4.2.5. Hypervisor based access control 44
4.2.6. Rootkit Detection Solutions 45
4.2.7. Hypervisor based monitoring and introspection 45
4.3. Adversary Model 46
4.3.1. Hardware 46
4.3.2. Hypervisor 46
4.3.3. Target VM 47
4.3.4. Trusted Service 47
4.4. Radium: Architecture Overview 47
4.4.1. ARTM 48
4.4.2. Hypervisor 49
4.4.3. Trusted and Untrusted Environments 50
4.4.4. Measuring Service 51
4.4.5. Access Control Policy Module 51
4.4.6. Working 52
4.5. Prototype Implementation 54
4.5.1. Boot-up 55
4.5.2. Protecting Confidentiality of Measuring Service 56
4.5.3. Launching a Measuring Service 56
4.5.4. Measuring the Target VM: 56
4.5.5. Attesting the Verification: 58

4.6. Performance Analysis 58
4.7. Security Analysis 59
4.7.1. Hardware 59
4.7.2. Hypervisor 60
4.7.3. Target VM 61
4.7.4. Trusted Service 61
4.8. Summary 61

CHAPTER 5 ONTOLOGY BASED VULNERABILITY MANAGEMENT FOR CLOUD COMPUTING 63
5.1. Introduction 63
5.2. Background 65
5.3. Related Work 66
5.4. Vulnerability Analysis Framework 69
5.5. Framework Implementation 72
5.5.1. OVM and OSAT 72
5.5.2. OVDB Framework 73
5.5.3. Ontology 73
5.5.4. Working 74
5.6. Vulcan 75
5.6.1. NVD 75
5.6.2. OKB 76
5.6.3. System Classifiers 77
5.6.4. Indexer 77
5.6.5. Vulnerability Class Index 77
5.6.6. SNLP 78
5.7. Vulcan Working Flow 78
5.8. Vulcan Implementation 81
5.8.1. OKB 81

5.8.2. Modules 83
5.8.2.1 System Classifiers 84
5.8.2.2 Indexer 84
5.8.2.3 Vulnerability Class Index 85
5.8.3. SNLP 85
5.9. Future Work 86
5.10. Summary 87

CHAPTER 6 SECURE AND TRUSTED EXECUTION FRAMEWORK 88
6.1. Introduction 88
6.2. Overview 89
6.2.1. Goals 89
6.2.2. Design Principles 91
6.3. Architecture 93
6.3.1. Hardware Layer 93
6.3.1.1 Immutable Root of Trust for Measurement (IRTM) 93
6.3.1.2 Processing Subsystem 94
6.3.1.3 Root of Trust for Storage and Reporting 95
6.3.2. Control Layer 95
6.3.3. Service Layer 96
6.3.4. Management Layer 97
6.4. Analysis 97
6.4.1. Implementation 97
6.4.1.1 Hardware Layer 98
6.4.1.2 Control Layer 99
6.4.1.3 Service Layer 99
6.4.1.4 Management Layer 100
6.4.2. Security and Trust 101
6.4.2.1 At the Time of Launch 101

6.4.2.2 Any Time During the Execution 102
6.4.2.3 At the Time of Change/Update 103
6.4.3. Challenges 103
6.5. Summary 105
CHAPTER 7 CONCLUSION 106
7.1. Contributions 106
7.2. Future Directions 107
APPENDIX: AUTHOR BIOGRAPHY 108
REFERENCES 111

LIST OF TABLES

Page

Table 4.1. Comparison table of various trusted computing solutions 60

LIST OF FIGURES

		Page
Figure 3.1.	Virtualization Based Secure Execution and Testing Framework	21
Figure 3.2.	Attack Model	26
Figure 3.3.	CTrust Cloud Framework	33
Figure 4.1.	Static Root of Trust for Measurement	40
Figure 4.2.	Dynamic Root of Trust for Measurement	41
Figure 4.3.	Adversary Model	43
Figure 4.4.	Radium Architecture	48
Figure 4.5.	Radium Prototype	54
Figure 5.1.	Vulnerability Assessment Framework	67
Figure 5.2.	OVDB Ontology	69
Figure 5.3.	OVDB Framework Search Page	71
Figure 5.4.	Vulcan Architecture	76
Figure 5.5.	Vulcan Working	80
Figure 5.6.	High Level View of our Vulnerability Ontology Definition	82
Figure 5.7.	VULCAN Modules	84
Figure 5.8.	Vulcan Semantic Natural Language Processor	85
Figure 6.1.	Secure and Trusted Framework - Architecture	94
Figure 6.2.	Secure and Trusted Framework - Implementation	98

CHAPTER 1

INTRODUCTION

1.1. Ubiquitous Computers

Computers have become an integral part of our daily lives. Computers are used in every aspect [84] of our lives from the morning alarm to bed time book reading. While some of the ways in which we use computers may seem mundane (for e.g., watching movies, playing music, instant messaging), other uses can have far reaching implications. For instance, we depend on computers for our banking needs; from receiving our wages to making purchases. Some other critical uses for computer applications can be found in the areas of aircraft flight control, weapons, nuclear systems [77] and delivering critical care medicine [49].

As our dependence on these computing systems continues to increase, so does the quantity and sensitivity of information stored and processed by them. This information includes confidential personal data like secret passwords, credit card numbers, and bank account numbers. As computing systems store and process increasing amounts of personal and sensitive data, they become lucrative target for malicious entities.

Even though we greatly rely on these computing systems for our daily needs and entrust them with our personal and sensitive information, the current state of their security and trustworthiness is not very reassuring. Pandalabs Annual Report [40] estimates there were a total of about 75 million samples or 285,000 samples of malware per day created in the year 2017 alone and this trend has been rapidly increasing in the recent years. Ransomware attacks like WannaCry, Petya, and NotPetya [16, 15, 14] have caused significant monetary losses and business disruption for individuals, governments, and businesses alike. These type of attacks have become so severe that the Ransomware 2017 [47] report by Symantec highlights how the ransomware is evolving in complexity similar to cyber espionage.

In recent years, high profile data breaches have become very common. It seems as if we are hearing about a major data breach as frequently as once every month. These data breaches have far reaching impacts as they compromise the identity and security of thousands and millions of users. In 2013, Yahoo Inc., has suffered the biggest data breach [17] in history. Around 3 billion user accounts have been compromised and 350 million dollars of losses were publicly disclosed. Later in 2016, an online dating website, Adult Friend Finder was breached and over four hundred million user accounts have been compromised [114]. In this attack, personal, confidential and potentially embarassing details about the customers have been made public. Further, in 2017, Equifax has faced one of the most critical security breaches [115]. This breach has result in the leakage of extremely sensitive and identity information of the customers. This included social security numbers, birth dates, and addresses of 147 million users.

For most, if not all software products, security is an after thought. This attitude along with the fact that there is no easy way to formally specify and evaluate software, makes them prone to many security vulnerabilities. The National Vulnerabilities Database [7] lists almost hundred thousand vulnerabilities with over twelve thousand discovered in 2017 alone. Not only are there more vulnerabilities being discovered than ever, they are also increasing in their severity level [13]. The presence of such vulnerabilities will only multiply the threats posed by malicious programs and actors.

With increasing malware attacks, data breaches, and vulnerabilities the outlook for the trust and security in computers is very bleak. Without proper security controls in place, the data can be stolen, modified, and held as a hostage for ransom. Thus computer security and trustworthiness has now become ever more important to avoid these systems from leaking out this critical information to unauthorized entities and enable smooth and seamless interaction with the technology solutions enabled by the computers.

1.2. A Vision for Trusted and Secure Computing Systems

In general, computer security aims at providing confidentiality, integrity and availability [120] with computing systems. Confidentiality is breached when information is accessible

to an unwanted and unauthorized entity. This entity can be a human, a software program or another computing system. The integrity is infringed when information is modified by an unauthorized entity and the availability is disrupted when a malicious entity succeeds in overloading the computing system with illegitimate requests. Availability may also be affected due to system failures or errors that are unrelated to security attacks.

Trusted Computing Group (TCG) [163] defines that a computing platform is trustworthy if it behaves as expected. To achieve this, they have proposed hardware and software based mechanisms. Using these mechanisms it is possible to attest the trustworthiness of a computing system to certain degree.

Many of the computer security and trust solutions try to address these challenges individually. While trust and security are orthogonal to each other, it is important to have a holistic view and approach for a computing system's security and trustworthiness. For example, a security solution is not very reliable if it is not trustworthy. In this dissertation we analyze various security and trustworthy solutions for modern computing systems. We then present our proposed framework that will provide holistic security and trust for the entire life cycle of an application.

1.3. Methodology

To provide holistic trust and security for a computing system, we have designed three novel techniques. These techniques have been developed and tested in virtualized environments providing trust and security for virtualized workloads. A virtualized workload is any independent unit of software executing on top of virtualized infrastructure.

This can be a full stack workload which includes an application, necessary libraries, and an operating system. Or it can be a lean stack workload which includes an application and optional libraries without an operating system. Since the workload can mean an application as well, we use the terms application and workload synonymously wherever applicable.

Given the proliferation of virtualization from cloud servers to embedded systems[126, 98, 122, 119], techniques presented in this dissertation, can be implemented on most comput-

ing systems. By using these three techniques, we will be able to provide trust and security in 3 phases of the application life cycle. These three phases represent the entire lifecycle of any application or workload in a virtualized environment. These phases are generic and can be applicable to regular applications as well. The three phases are:

(1) At the time of launch

(2) Any time during the execution

(3) After a change or update

In this dissertation, we investigate different trust and security requirements in these three stages and provide a comprehensive framework that allows secure and trusted execution of the workload throughout its lifecycle. Further, this dissertation focuses only on confidentiality and integrity of the virtualized workload. We consider that availability of the computing system is out of the scope of this dissertation.

1.3.1. Secure Execution

The most important security feature necessary for a computing system is secure execution. This feature will allow a computer program to execute with its integrity or confidentiality uncompromised. Without security guarantees, it is pointless to prove that a computing system is trustworthy. Hence we will first focus on how to create secure execution environment for virtualized workloads and then focus on how to provide trust in this securely executing workloads.

There are software and hardware approaches to provide secure execution environment and guarantee the secure execution of a computer program. While the hardware approaches usually come with more guarantees, they are very difficult to build, test, and manufacture at necessary volumes. By the time a hardware architecture is designed, thoroughly tested, and released into the market, it is usually outdated to the latest requirements. These challenges make hardware approaches difficult to embrace in the real world outside of the academia and research communities.

Traditionally, operating systems are responsible for providing a secure execution environment. With millions of lines of code, the operating systems are potentially vulnerable to many attacks and are hardly trustworthy. To provide an effective and trustworthy secure execution environment, the provider of the secure execution environment needs to have smaller Trusted Computing Base (TCB), should be resilient to powerful attacks (e.g. attacks launched from the operating system context), and allow fine grain controls of what objects can be secured.

In Chapter 3, we present a virtualization based secure execution solution that fulfills the above requirements:

(1) Secure execution is provided by hypervisors which have few thousands of lines of code (smaller TCB) and are naturally less vulnerable and more trustworthy.

(2) Is resilient against attacks launched by powerful adversary, such as a compromised and/or malicious Operating System (type 1 hypervisors execute at a higher privilege level).

(3) Allows page level granularity in providing security to the necessary components of the secure process.

1.3.2. Trust and Integrity Measurement

It wouldn't be just enough to ensure that a computing system is executing securely. Like aforementioned, we need to provide guarantees that a securely executing system should also be trustworthy. A system will be trustworthy when we have verifiable proof that it is executing code that we 'know' or 'trust' to be secure. It is important to understand that we are only focusing on providing trust guarantees to a system to ensure that we are indeed executing what we know or trust to be good/benign/secure. Such a trust guarantee is usually provided as hash measurements which would indicate the state of the computing system.

To provide trustworthiness in a computing system, we have to ensure that an application is launched in a trusted state. In other words, the application is in a state we trust to be good/benign. To provide such a guarantee, we will have to make sure every software component that is launched prior the application of interest is also trusted. This will guar-

antee that the application is indeed launched in a trusted environment. To provide such a guarantee, we can use the chain-of-trust as demonstrated in SRTM [36] and DRTM [66]. Both these methods start with a hardware based chip [123] as root-of-trust that cannot be modified or tampered with.

After ensuring that the application is launched in a trusted state, there will be many hardware and software components interacting with the application. It is possible to ensure that these interactions may alter the state of the system and render our initial trust in the system invalid. This would raise questions about the trustworthiness of the system at any point after the launch. How can we guarantee that the system is executing in trusted state at any time after the launch? Would such guarantees be trustworthy without the chain-of-trust? Can these guarantees be trustworthily reported to a remote or local verifier?

To address these questions, Chapter 4 presents the Radium architecture [80]. By extending the hardware root-of-trust to the hypervisor, Radium facilitates asynchronous, on-demand integrity measurements that are more flexible and granular.

1.3.3. Vulnerability Management

While we provide a secure execution environment and on-demand integrity measurements to ensure that a system is running in a secure and trusted state, we should be vigilant of potential bugs and vulnerabilities in the system. As discussed earlier, there is no easy way to specify and verify a software. To tackle this problem is out of the scope for this dissertation. Instead, we provide a system to analyze and manage the vulnerabilities or defects in the system to ensure that the system is free from any known vulnerability and thus continues to be in a secure state.

As modern systems have become complex with multi-million lines of code along with hundreds and thousands of dependencies, it is a huge challenge to manage vulnerabilities. Especially when these systems are linked with many libraries, dependencies, and other systems. To exacerbate this increased complexity in the systems, vulnerabilities have been rising in both number and severity. To tackle this challenge effectively, we have to organize and manage vulnerabilities in an efficient way. We should also be able to provide a way to

search, analyze, and query these vulnerabilities in a natural and intuitive way.

To tackle these challenges, in Chapter 5 we present an Ontology based Vulnerability Database (OVDB) [79] and a Vulnerability Assessment Framework for Cloud Computing [72] based on OVDB. Ontologies [51] are used for sharing knowledge and can model real world concepts, such as vulnerabilities, which cannot be efficiently organized as taxonomies. With their roots in knowledge modeling and artificial intelligence, ontologies also make it easier to semantically reason and analyze the vulnerabilities.

1.4. Summary

During the course of investigating principles to design a holistic trusted and secure computing system to ensure confidentiality and integrity throughout various stages of an application's lifecycle, this dissertation makes the following contributions:

(1) A novel technique to securely execute an application even in the presence of malicious operating system using hardware virtualization.

(2) A new integrity measurement architecture that extends the hardware root-of-trust to the hypervisor to facilitate on-demand measurements that are more flexible and resilient to TOCTTOU attacks.

(3) A comprehensive framework for organizing, managing, and assessing vulnerabilities in complex computing systems such as cloud using ontologies.

(4) A set of guidelines that can be used to design and build a secure computing system that can be verified for trust guarantees at any time of the system's lifecycle.

CHAPTER 2

BACKGROUND AND MOTIVATION

Instead of starting from scratch, the solution presented in this dissertation builds on top of many existing technologies and solutions. To understand the design decisions and effectiveness of the presented framework, it is important to have insight into the technologies we rely upon. In this chapter, we will discuss technologies that have laid the foundation for our solution. We will also highlight the limitations with these technologies that have motivated us to design our solution. These discussions have been organized thematically as follows:

Section 2.1, discusses some of the hardware based architectures designed to provide secure execution for applications. We will discuss how the knowledge of these architectures provide us the key features necessary to build secure execution environments. Later, in Section 2.2, we will discuss the advancements in virtualization technology and how it presents a solid foundation to build modern day security and trusted computing solutions. In Section 2.3, we will introduce trusted computing and discuss how the modern day trusted computing solutions can be built without any expensive hardware and significant modifications to the commodity hardware. Finally, in Section 2.5 we will present some approaches to vulnerability assessment.

2.1. Hardware Security Architectures

Hardware security architectures typically consist of modified hardware components and/or configurations to provide security to the application. These hardware based security solutions differ quite a bit in their approaches in what hardware components need to be modified. Similarly, they also differ in what software should be part of the trusted computing base. Some architectures include Operating System as a part of the trusted computing base while some architectures exclude the Operating System from the trusted computing base.

Unlike software, which is dynamic in nature, hardware will not alter its state or function. This property of the hardware in general makes it ideal choice for tamper resistant

solutions. Thus, hardware security architectures are usually geared to provide solutions where confidentiality and integrity are critical, for example, to provide Digital Rights Management (DRM).

Solutions such as Citadel [153] and ABYSS [152] provide a physically tamper-resistant package with the entire processing subsystem encapsulated within this package. Such a subsystem usually contains FLASH ROM, CPU, DRAM, and BBRAM [111]. This subsytem is responsible for executing the security sensitive portions of the application. These solutions, along with Dyad [160] execute only signed code from trusted entities. The trusted hardware contains the private key of a public and private key pair which is used to authenticate applications to remote verifier.

Although AEGIS [128] architecture has goals and functionality similar to the above secure co-processor solutions, it uses a design similar to XOM [159]. Both these solutions execute secure applications in an isolated secure compartment which cannot be accessed or interfered by regular applications. This is achieved by encrypting data and instructions of the secure applications and the decryption key is protected by the hardware. The secure applications are decrypted only in the secure compartment. While XOM requires all instructions in the secure compartment to be encrypted, AEGIS uses encryption and integrity verification whenever required.

Another security architecture HIDE [162] encrypts cache entries to obfuscate memory access patterns so that it is impossible for an attacker to identify the Control Flow Graph (CFG) and by extension software libraries used in the application. When such a control flow graph is identified for well known open source libraries used in the secure application, they can be exploited to compromise the security of the application. In Arc3D [56] it was shown that operating systems cannot be trusted to provide necessary security guarantees for the applications with DRM requirements. Further, it was recommended that the underlying architecture should be providing these security guarantees to the applications.

Similar features are also provided by the IBM 4758 [124] cryptographic coprocessor based solutions. In this architecture, the system contains a regular microcomputer processor

(Intel 486), a special chip for cryptographic operations, and memory subsytem in a tamper resistant secure package. By using a BBRAM, the package can actively respond to any physical attacks to tamper the trust and security of the system. When any such attacks are detected, all the sensitive information is deleted and the device will be permanently disabled ensuring the confidentiality of the sensitive information.

From the above discussion in hardware security architectures, we can summarize the following observations: Hardware provides a tamper resistant root-of-trust to build solutions on top of. Only security sensitive portions of the application need to be protected and operating systems are not trustworthy to provide these security guarantees.

2.2. Virtualization

In Computer Science, virtualization refers to abstraction or emulation of hardware in the software layer. In this dissertation, we refer to virtualization with respect to the emulation and/or simulation of hardware machine as described by Goldberg [55]. Such a simulated machine is called as Virtual Machine and the software doing the simulation is called Virtual Machine Monitor (VMM) or Hypervisor [118]. Virtual machines and virtualization have gained lot of mainstream adoption in the recent years due to their benefits in resource consolidation and efficiency. Cloud Computing [75, 95] has been the biggest proponent of this widespread adoption of virtualization.

As discussed in the previous Section 2.1, hardware security architectures are modifications to the hardware to provide superior trust and security to the applications. As virtualization simulates the hardware components, it provides a more flexible platform to implement security architectures. In the *Framework for Design Validation of Security Architectures*, Dwoskin et al. [48] shows that hardware security architectures can be implemented in the virtualization layer.

Virtual Machine Monitor (VMM) and Hypervisor are used interchangeably while Virtual Machine (VM) can also be referred to as guest, domain, and partition. Hypervisors may have their own device drivers to allocate and control the hardware resources. Alternatively, they can rely on other operating systems to provide divers for the hardware resources. De-

pending on how the hypervisor interacts with the underlying hardware, the hypervisors are typically classified into two types [44]:

(1) Bare Metal

A Type 1 hypervisor or a bare metal hypervisor executes directly on the hardware and is typically loaded first in to the memory by the bootloader. A popular example for this type of hypervisor is Xen [26]. A type 1 hypervisor may use one of the virtual machines a privileged guest and uses it to access the hardware resources. In Xen hypervisor, such a privileged guest is referred to as Domain Zero (Dom0). In contrast, VMware [117] and KVM [58] have a monolithic architecture that bundles the device drivers along with the hypervisor.

A virtualization aware guest is often referred to as ParaVirtual (PV) machine and contains drivers that communicate with virtualized hardware rather than the physical hardware. A guest unaware of it being virtualized is referred to as Hardware[-assisted] Virtual Machine (HVM) and often consists of unmodified Operating System executing on top of it. In this dissertation, we focus on the type 1 hypervisors as they have complete control over the physical hardware resources and more suitable for implementing security architectures.

(2) Hosted

A Type 2 hypervisor or a hosted hypervisor is executed by the Operating System as an application and relies on the host Operating System' drivers for hardware access. It is important to understand that the type 2 hypervisor do not have absolute control over the entire physical machine's hardware. Rather, type 2 hypervisor only has control over the resources allocated to it by the host operating system. However, the type 2 hypervisor can exercise complete control over the resources allocated by it to the guest virtual machines. VirtualBox [150] is a good example of type 2 hypervisors. In functionality, type 2 hypervisors are much similar to the programming language virtual machines such as JVM [97, 141].

2.2.1. Hardware Support for Virtualization

With the increase in the adoption of virtualization technology, hardware vendors started implementing support for virtualization [138] within the hardware. These are commonly referred to as Virtualization extensions. We will discuss the 2 major platforms and their support for virtualization.

(1) Intel

Intel has introduced hardware support for CPU virtualization (commonly referred to as Intel VT-x) that provides instructions to switch the processor operation between VMX root mode (where the hypervisor gets control) and VMX non-root mode [136, 104]. This support allows the processor to reduce the number of instructions that need to get trapped while improving performance and complexity of the VMM code base. Intel has also support for memory virtualization through Extended Page Tables (EPT) which allow additional indirection for the VM physical addresses. This improves the performance of in-guest memory lookups and accesses by not involving the hypervisor. Finally, through I/O device virtualization (commonly referred to as Intel VT-d) it becomes trivial for multiple virtual machines to share I/O devices in a secure and efficient manner [54].

(2) AMD

Similar to Intel, AMD also support CPU, memory, and I/O virtualization. The AMD Secure Virtual Machine (SVM) [138, 19] technology provides the ability to selectively trap sensitive instructions and use a Virtual Machine Control Structures (VMCS) to schedule virtual machines as processes by the VMM kernel. With the use of Nested Page Tables (NPT) [46], AMD allows the virtual machine's operating system to allocate and control its own memory. A VM can also request for dynamically increasing and decreasing the memory allocated to it by the VMM. With the introduction of I/O Memory Management Unit (IOMMU), AMD has support for I/O device virtualization [23] and provides the guest OS to manage its own I/O device interactions. Alternatively, there can be a dedicated driver domain that acts

as a proxy for all virtual machines' access to the I/O devices [146].

To summarize, modern hardware virtualization technology can provide an efficient and secure abstraction of the underlying hardware devices to the VMM. This also gives great control over the hardware resource allocation and allows the VMM layer to decide which instructions need to get trapped into the VMM layer. This allows the VMM code to be simple and efficient while giving us the ability to modify and control any hardware behavior exposed to the virtual machine by the hypervisor. As the hypervisor runs at a higher privilege level than the guest operating system, we have hardware enforced protections for the VMM. Leveraging these features, secure [hardware] architectures can be implemented in the virtualization layer.

2.3. Trusted Computing

For the purpose of this dissertation, we define trusted computing as "A computer system for which an entity has some level of assurance that(part of or all of) the computer system is behaving as expected" [99]. The nature of this entity and the degree of the assurance can vary greatly from system to system and component to component.

To facilitate trusted computing in modern computers, Trusted Computing Group (TCG), a consortium of computing companies, has designed the Trusted Platform Module (TPM) that provides a secure root-of-trust to store and report the system measurements [100, 154, 129]. TPM provides attestation service, storage service and endorsement service for the platform. Programs that use cryptography keys to secure user data can use TPM to build more trustworthy applications as hardware based cryptography is immune to lot of software based attacks. TPM is a generic solution with platform specific implementations. In this dissertation, we limit our discussion to the PC platform specific implementation of TPM commonly found in desktops, laptops, and servers.

TPM consists of Platform Configuration Registers (PCR). A PCR is a 160-bit register with special properties. The values stored in PCRs can be used to verify the measurements. The PCR values are used to define the state of the system. Various software components running on the system are measured and these measurements are stored in the PCRs. A single PCR or a set of PCR values is used to represent the state of the platform. By extending

PCRs with the measurements of various components in the system, the state of the platform can be recorded. This recorded state of the platform can later be used to verify if the system is in a trusted state.

In PC Platform there are a total of 24 (0 ... 23) PCR registers in TPM (version 1.2). PCRs 0–4 store measurements related to BIOS, ROM, and Memory Block Register. PCRs 5–7 store measurements related to OS loaders. PCRs 8–15 are known as static PCRs that store measurements related to OS. These are typically used to record and verify the state of the system during the boot time. PCRs 17–22 are known as dynamic PCRs and are used to store measurements of the system dynamically. These are used either at the boot time or at any other time as necessary. Some trusted computing solutions such as SRTM and DRTM use these PCRs for verifying the platform state at boot time and at a dynamic reset.

The TPM has several cryptographic keys and keypairs. These keypairs, along with the PCRs, are used to perform operations to store, measure and verify the system state. TPM uses the following special operations [123] that makes the storage, measurement, and verification tamper resistant.

2.3.1. Extend Operation

PCR is designed in such a way that the only write operation it supports is *Extend*. In other words, a PCR can not be assigned a specific value using the write operation. This makes the PCR non-commutative and allows it to store infinite number of measurements. After an entity is measured, to be stored in the PCR, it is hashed with the PCR's existing value. If a PCR has to be extended with a measurement value X, the following operation is performed by the TPM.

$$\mathrm{PCR}_{\text{new value}} = \mathrm{Hash}(\mathrm{PCR}_{\text{existing value}} || \mathrm{X})$$

2.3.2. Bind Operation

Using the TPM, we can *bind* any secret to the platform and save it to the TPM by encrypting the secret with a *storage key*. This storage key is an asymmetric encryption key and the *secret* is bound to the platform by encrypting the *secret* with the public part of the

storage key. The private part of the *storage key* never leaves the TPM hence guaranteeing that only the TPM which has access to the *private key* can decrypt the secret. Since these storage keys are unique to each chip, we can be confident that a secret bound to a particular TPM, can only be decrypted by that particular TPM.

2.3.3. Seal Operation

Sealing adds platform state to the binding operation. The value being stored is not only associated to a particular TPM but is also associated with the values stored in the PCRs. In other words, if a value is sealed with specific set of PCR values, it can only be unsealed when that particular set of PCRs have the values expected at the time of sealing, implying the platform is a known state.

2.3.4. Quote Operation

The TPM allows verification of the platform state through the Quote operation. A verifier can request a quote of the platform state from the TPM. The TPM will issue a quote, hash of set of the PCRs signed by the TPM's AIK. The quote has the PCR values that will attest the platform state. The quote operation is used to ensure the platform is in a trusted state before running security critical applications or to verify that an application of interest has executed as expected.

With its tamper resistant design, cryptographic keys, and secure operations TPM can be used to build trustworthy systems using modern computers with little-to-no modification. in fact, many modern computing systems come equipped with the TPM out of the box. It is also important to note that the TPM functionality is designed to be generic for any platform and is very inexpensive. This allows TPM to be integrated into any kind of device and can be expected to be widely available. This makes the TPM an ideal root-of-trust to build a trusted computing solution upon.

2.4. Cloud Computing

Cloud Computing [43, 103] is essentially a new information services delivery model that has gotten widespread adoption in the recent years. It can be defined as a model for

delivering computing services such as storage, computation power, network bandwidth, and software across the Internet using virtualization [158, 88, 83]. This has been a significant paradigm shift in how information services are delivered and has become the de-facto standard to deliver such services for end-users and enterprises alike. The delivery of the services in Cloud Computing can be classified into 3 categories [94].

2.4.1. IaaS

In the *Infrastructure as a Service (Iaas)* [29] delivery model, hardware or infrastructure related computing resources are provided on-demand as a service. These service primarily include:

(1) CPU/Processor time

(2) RAM/Memory

(3) Disk Storage

(4) Network Bandwidth

The IaaS is the most fundamental offering of the Cloud Computing paradigm. This service enables users to subscribe to the hardware resources whenever they want with much finer granularity. It also prevents them in investing in the hardware that they may not use to the full extent, or sometimes, not use at all due to change of plans. The IaaS also allows the users to scale up or scale down their infrastructure with great flexibility and very little overheads or delays. Some of the major IaaS offerings are: Amazon Web Services, Microsoft Azure, Google Cloud, and Digital Ocean. An IaaS user is responsible for the security of the hardware resources and anything that executes on top of it for e.g., operating systems, application platforms, and applications.

2.4.2. PaaS

After the IaaS, *Platform as a Service (PaaS)* [109] is the most important offering. Instead of the raw hardware resources, PaaS provides a ready-to-use platform for running applications. This allows the developers to focus on developing and deploying the application and not worry about the provisioning and administering the hardware resources. PaaS

offerings provide APIs to integrate into the user's development process, load balancing, and tools to scale the application dynamically. One key advantage of PaaS over the IaaS is that it is more easier to use and has better interoperability for the user across cloud providers. Some of the notable PaaS offerings are: Google App Engine, Heroku, CloudFoundry, AppScale. A PaaS user will be responsible for the security of the application and the user data stored by the application.

2.4.3. SaaS

Software as a Service (SaaS) [45] is probably the most popular model of delivering cloud services. SaaS refers to an application provided as a service and billed through pay-as-you-go model. Unlike the traditional product delivery model where the user buys a product and owns it forever. SaaS model is very convenient model for offering new services as there is no overhead on the end-user's part other than signing up for the service. Some of the notable SaaS applications are: Google Apps, Salesforce, Workday, Concur, and Cisco WebEx. A SaaS user is responsible only about the security of their own account when trusting the security of the application with the service provider.

While Cloud Computing makes delivering information services very convenient, it also complicates the security challenges to the cloud providers and end users alike. Storing data and executing applications in remote locations administered by the cloud provider reduces visibility and control for the users. Cloud providers have to deal with unknown code executing in their systems while protecting users from internal and external threats. It is often complicated by various layers introduced by the cloud computing. This typically includes: the hypervisor, operating system, platforms and/or libraries, applications, and management interface. To tackle this challenge in Cloud Computing or any virtualized environment, one would need a robust vulnerability management program.

2.5. Vulnerability Management

According to the ISO 27002 [68], a vulnerability is defined as "a weakness of an asset or group of assets that can be exploited by one or more threats". This weakness makes systems

vulnerable to threats and compromises the security of the system. Vulnerability management involves more than just scanning for vulnerabilities. It also involves risk evaluation, risk acceptance, and remediation [110]. It is extremely important to have a robust vulnerability management program to alleviate potential risks that arise due to the vulnerabilities.

NIST recommends [93] that every organization have a dedicated Patch and Vulnerability Management Group to monitor, evaluate, and remediate vulnerabilities within the organization. NIST also provides several databases, tools, techniques, and processes for managing security vulnerabilities [3, 7, 5, 2, 4, 60, 25]. Gartner lists vulnerability management as one the most critical security measure [85] for Cloud Workloads. Despite vulnerability management being an important aspect of any organizations' security arsenal, current practices are not very promising.

With the accelerating adoption of cloud computing and rapidly changing IT landscape, vulnerability management becomes even more challenging. To properly address these challenges, it is imperative to use standards based solutions. For this purpose, NIST recommends using Ontologies and even developed an Open Vulnerability and Assessment Language [25] for security content and automation. By using ontologies, we can provide effective vulnerability management and automation.

CHAPTER 3

VIRTUALIZATION BASED SECURE EXECUTION[1]

To provide security and trust in a computing system, it should at least guarantee the secure execution of the applications. This becomes even more critical in case of remote or distributed computation. In this chapter we will introduce the challenge of secure execution, historical approaches and limitations. We will then present our virtualization based secure execution framework along with an implementation in the context of cloud computing.

3.1. Introduction

The recent years have seen a wide scale growth, adoption and popularity of the Virtualization Technology [135] to provide efficient and cost-effective usage of expensive hardware. Virtualization technology introduces a software abstraction layer or virtualization layer (virtualization software) between the hardware and the operating system, thus decoupling them from each other. This software abstraction layer is known as a Virtual Machine Monitor (VMM) [121] or the hypervisor. A VMM / hypervisor allows the user to create multiple Virtual Machines on a single physical hardware platform, each capable of running an operating system (O.S.) and its applications. Virtualization provides security by isolating the guest O.S and its applications in a single virtual machine. Thus any security failure in a particular guest OS does not affect the functioning of other guest OSs running on the system.

A virtualization software emulates the underlying hardware platform to provide a known interface for the OS and applications to work on. This makes it much easier to incorporate hardware security mechanisms within the virtualization layer as compared to on-chip. Moreover the performance overheads incurred by the virtualization softwares are significantly low as observed by Younge et al. in [161]. Thus the security of the entire system can be increased without accounting for significant performance overheads.

[1]Majority of this chapter has been previously published from S. Kotikela, S. Nimgaonkar, M. Gomathisankaran, International Association of Science and Technology for Development (2011). Reproduced with permission from ACTA PRESS

With this motivation, we propose a Virtualization Based Secure Execution and Testing Framework by modifying an open source Virtualization Software - Xen [26]. Our framework provides a generic interface to plug-in an existing secure architecture. Once plugged-in, the attack suite in our framework performs a series of attacks on a secure application/process running on the underlying secure architecture. A log of all these events is stored which can be later reviewed to judge if the secure architecture behavior is secure or not.

The rest of the chapter is organized as follow. Section 3.2 describes our Proposed Framework. Section 3.3 presents the Implementation of our framework. Section 3.5 describes the Related Work followed by the Future Work in Section 3.6 and Summary of contributions in Section 3.7.

3.2. Proposed Framework

The proposed Virtualization Based Secure Execution and Testing Framework for testing hardware secure architectures is developed on top of Xen Hypervisor [26], an open-source Virtualization Software. Therefore our framework uses some of the components already provided by Xen. Figure 3.1 below shows all the components of the proposed architecture along with their functioning. These components are Xen Hypervisor, MoCo VM, Application VM, Event Trigger Mechanism, Secure Architecture Plug-in Interface (SAPI) and Attacker VM. The MoCo VM in turn consists the Monitor and the Controller while the Application VM consists the Secure Application running on top of the underlying Secure Architecture.

3.2.1. Xen Hypervisor/VMM

The Xen Hypervisor is the basic abstraction layer (virtualization layer) software that sits directly on the hardware below the operating system. It emulates the underlying hardware and is responsible for CPU scheduling and memory partitioning in order to allow multiple Virtual Machines to run on single hardware platform. Xen Hypervisor controls the execution of all the Virtual Machines running on it, however it has no knowledge of networking, external storage devices, video, or any other common I/O functions found in a computing system.

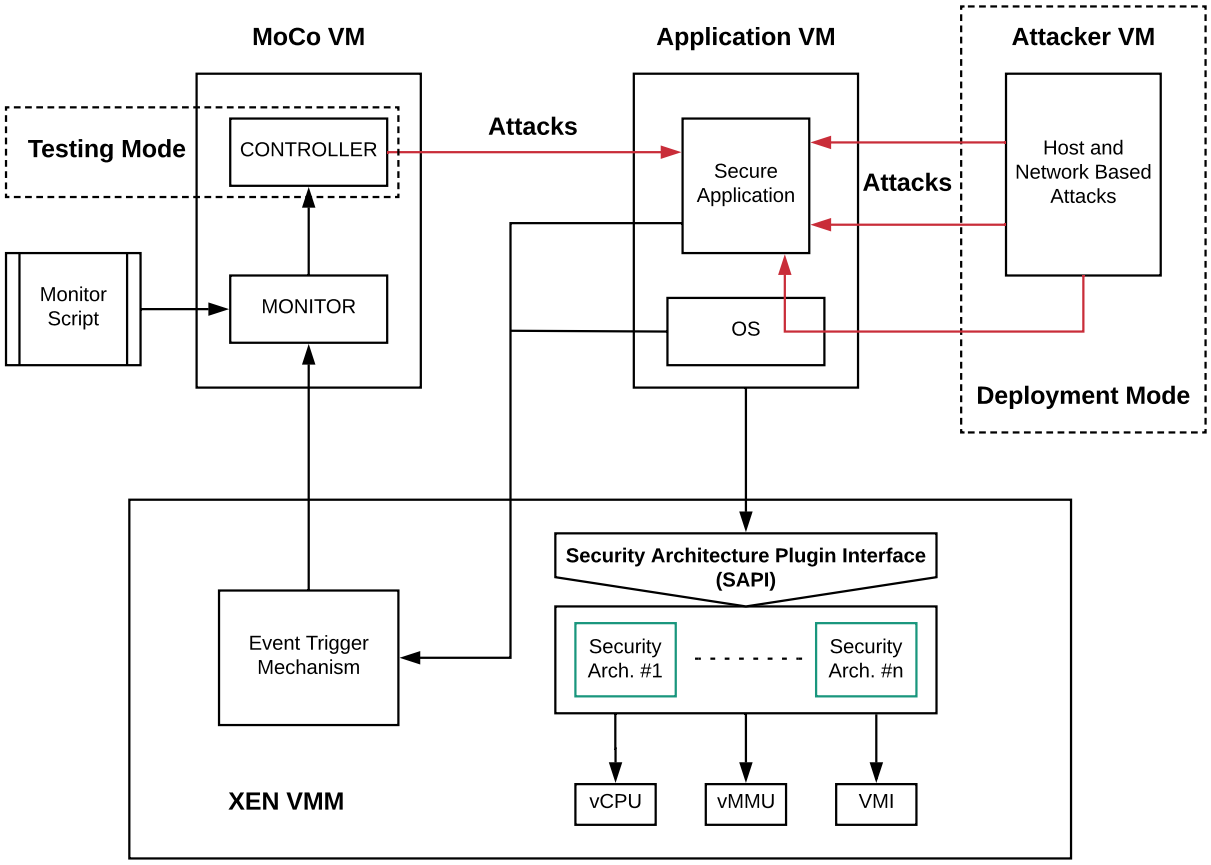

FIGURE 3.1. Virtualization Based Secure Execution and Testing Framework

3.2.2. MoCo VM

A MoCo VM - Monitor/Controller Virtual Machine is essentially a specialized DOM 0 kernel found in the Xen architecture. DOM 0 also known as Domain 0, is a specialized Virtual Machine running on the Xen Hypervisor with special privileges to access physical I/O resources and communicate the other Virtual Machines running on the system. In its simplest form, a DOM 0 is a modified Linux kernel, that must be running on all Xen Virtualization environments before any other Virtual Machines can be started. In our framework, the DOM 0 kernel contains the Monitor and the Controller to form a MoCo VM.

3.2.3. Application VM

An Application VM is a DOM U kernel found in the Xen architecture. DOM U also known as the Domain U, is an unprivileged Virtual Machine running on Xen Hypervisor. Xen currently supports both para virtualization and full virtualization. In para virtualization, the guest kernel has to be modified in order to run on Xen e.g. Linux OS, Solaris, FreeBSD et. Virtual Machines running such a kernel are known as DOM U PV Guest. Whereas in full virtualization, the guest kernel is not modified e.g. Windows. Virtual Machines running such a kernel is known as DOM U HVM Guest. In the proposed framework, a DOM U kernel is used to run the Secure Application that utilizes the security mechanisms provided by the underlying plugged-in secure architecture.

3.2.4. Event Trigger Mechanism

Event trigger provides a mechanism to initiate inter virtual machine communication. Traditionally Xen provides some mechanisms and tools like split driver, xenstore, grant table and ring buffers to carry out inter VM communication. However the primary drawback of these mechanisms and tools is that they need the support from DOM U kernel to initiate communication. The proposed framework assumes all DOM U kernels to be untrustworthy and vulnerable to attacks. Hence we have avoided the use of these traditional mechanisms and have developed a new inter VM communication mechanism through the use of hypercalls and virtual interrupts (VIRQs), that are not dependent on the DOM U kernel. A hypercall works similar to a system call in a kernel or an OS. It is an interrupt typically *INT 81h* in Xen used to switch the control between the kernel and the Xen hypervisor. Hypercall interrupt the processor and are trapped in Xen using Hypercall Handler functions. VIRQ is a software interrupt notified by setting up a bit in the Virtual CPU data structure present in Xen.

3.2.5. Secure Architecture Plug-in Interface

The SAPI is a generic interface exposed by our framework to easily interface a secure architecture. SAPI is a collection of APIs which are used to modify virtual CPU (VCPU),

virtual memory management unit (VMMU) and provide memory introspection functions into the modified components of the Xen. Secure architectures are typically changes in CPU and memory of the hardware which facilitates secure execution and isolation of a process. These components are available as software modules in Xen, it would be easy to modify them and program (or change) according to different secure architectures. As shown in Figure 3.1, the SAPI primarily has three components: modified VCPU API, modified VMMU API and modified memory introspection API. The modified VCPU API has all the set of functions which modifies the Virtual CPU provided by Xen. These modifications include providing encryption and decryption capabilities to the processor, add extra and secure registers, modify or implement new cache memories etc. The modified VMMU has set of functions which enhance the existing virtual memory management Unit provided by Xen. These changes include switching on/off the conventional virtual memory layout of the operating system, encrypting and decrypting functions for main memory and adding additional access restrictions to the memory access of secure process memory pages. Though there are memory introspection functions already available by the Xen VMM, these threats are invalidated once security architecture is invoked. Hence new set of newly customized/modified memory introspection functions are needed for each secure architecture.

3.2.6. Monitor

Monitor is a program running in the DOM 0 that receives notifications about the Secure Application through VIRQs. It is responsible for monitoring critical events pertaining to the Secure Application and detect any possible software attacks on it. It contains a monitoring script which is a collection of watch events and actions to be performed when a particular event has occurred.

3.2.7. Controller

The security architectures should be rigorously tested against various attacks. These attacks are almost similar for all the architectures. Hence we used a collection of such attacks called the Controller. The Controller is a collection of host and network based

attacks launched from the DOM 0. While a secure application is running in the Application VM, controller mounts attacks on that application. A detailed attack model in presented in Section 3.2.11.

3.2.8. Secure Application

Secure Application is a special process whose memory is required to be protected from attacks on the system. This process is aware of the security mechanisms provided by the secure architecture plugged in to our framework. Secure Applications or processes protect their confidential data by storing them in specialized protected memory regions.

3.2.9. Attacker VM

Similar to the Controller, the Attacker VM is used to launch inter VM attacks on the secure application running in the Application VM.

3.2.10. Framework Functioning

The first step of the functioning is to plug-in a secure architecture by using the generic interface provided by the SAPI. Once this is done, the VCPU, VMMU and virtual memory introspection functions are appropriately modified to align with the plugged-in secure architecture. All the information pertaining to the security mechanisms provided by the secure architecture is then reported to the Monitor through the Event Trigger mechanism. The Monitor thus now has complete visibility and understanding of the secure regions within the secure architecture. Now the Secure Application can start executing in the Application VM. Typically, the entire memory allocated to the Secure Application need not be confidential and hence has no need to be protected. A Secure Application identifies its critical confidential data and then transitions to secure execution state.

In the proposed framework, this is achieved by invoking a secure hypercall *enter_vbase*. Once the Secure Application invokes this hypercall, the Secure Architecture and the Monitor are informed about the secure execution of the application. Its now the responsibility of the Secure Architecture to protect the confidential data of the application, while the Monitor is responsible for tracking all the necessary events related to the application. While the

application is executing securely, the Monitor informs the Controller to conduct a series of attacks on the secure application. During this time, the Attacker VM is also initiated to carry out inter VM attacks on the secure application. The secure application can exit secure execution by invoking the *exit_vbase* hypercall. At this instant, the Monitor stops tracking the events while the Controller and Attacker VM stop the attacks on the application. Once the secure application terminates, all the events recorded by the Monitor are available for review in a log file. This file serves as a benchmark to judge the effectiveness of the security mechanisms provided by the secure architectures.

3.2.11. Attack Model

In practice Secure Applications are modified to directly communicate with the secure architectures. Ideally this communication should by pass the standard communication mechanisms used by other applications, i.e. through the operating system. The primary reason being these other applications and the OS may have vulnerabilities that can be exploited by an adversary and hence are untrustworthy.

Instead of rooting the trust of Secure Applications in the OS, it is rooted in the secure architectures plugged into the framework. The software components in a computing system are susceptible to attacks like spoofing, splicing and replay. In a spoofing attack, an adversary successfully masquerades as an authorized entity to intercept valid requests from another authorized entity and returns a faulty or a malicious response to that request. A splicing attack is the one in which an adversary intercepts valid requests from an authorized entity and returns a valid but an unwanted response to the request. And finally in a replay attack, an adversary returns stale copies of responses to the requests made by authorized entities.

Figure 3.2.11 shows the generic threat model considered for the proposed architecture. Here the Virtualization Layer, Xen VMM containing the secure architecture and the Secure Application are two trusted entities, whereas the OS i.e DOM U kernel and other applications are considered to be untrustworthy. The major attack sources are network attacks, spoofing attacks, splicing attacks and replay attacks. These attacks occur on Secure Applications and

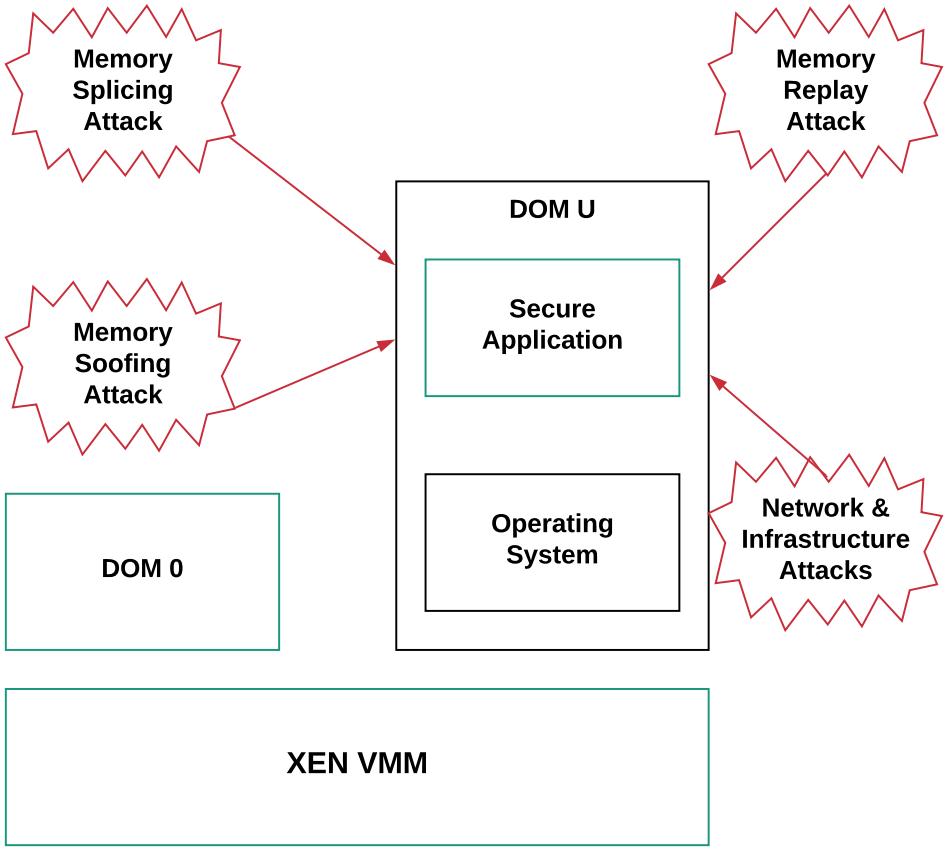

FIGURE 3.2. Attack Model

the untrusted entities in the system.

3.3. Implementation

We have implemented a secure architecture within the Xen VMM. This secure architecture provides two hypercall functions *enter_vbase* and *exit_vbase* to enter and exit secure execution. A hypercall handler is used to encode security architecture functionality. More robust and sophisticated security architectures can be done as separate modules (security components) with in the VMM. In any application a lot of code is derived from publicly available libraries. This code is not sensitive or critical from security perspective. Hence, most of the code need not be secured.

In the implemented example, we assume that secure process allocates a chunk of

memory. This chunk of memory is used to store all the security critical data. This is very similar to the heap allocated and maintained by JVM. The framework protects this chunk of memory using the implemented secure architecture. For ease, we have created the memory in the size of a page and aligned it to the beginning of the page boundary.

Algorithm 1 Secure Process Execution

1: Secure Process Begins
2: Allocate Secure Memory
3: The virtual address of secure memory is passed through ring3_hypercall
4: Virtual address of secure memory is received in hypervisor
5: Guest Frame Number (GFN) is computed for the received Guest Virtual Address (GVA)
6: Machine Frame Number (MFN) is computed for the GFN
7: MFN is made read-only
8: Virtual Interrupt (VIRQ) is sent to DOM 0
9: VIRQ is received in the DOM 0 kernel
10: DOM 0 kernel sends signal to the Monitor
11: Monitor prints messages to user

The working of the secure process in conjunction with the security architecture is described in Algorithm 1 followed by code explanation. It is important to emphasize that the prototype implemented in this paper is just a small instance of the framework. This prototype serves as a proof of concept to prove the effectiveness of the framework.

This implementation includes a very simple hardware secure architecture functionality and not a fully functional architecture. The implementation is not as powerful as the framework itself as its primary goal is to show the readers how an application can be secured in a virtualized environment. The secure application runs in a HVM DOM U. Once the secure process starts execution, it allocates a chunk of memory for secure data/variables. Later the secure process will allocate memory out of this secure chunk of memory to the secure data/variables.

The implemented security architecture uses vBASE to secure this chunk of memory. To achieve this, the secure process copies the virtual address of the secure chunk of memory and passes it to the hypervisor. The size of the secure chunk of memory is aligned with that of the memory pages. It is relatively easier to mark the pages read-only than making individual memory locations read-only.

The primary advantage of our implementation is that, it eliminates the Operating System in making the secure process's memory read-only. To achieve this, it uses the CPUID instruction. Because all other instructions except CPUID are short circuited to the operating system's interrupt table and do not trap in to the VMM. CPUID, on the other hand, will trap directly into the VMM. CPUID is originally intended to report processor's features to ring-3 software applications. It takes various parameters and reports different features of the processor depending on those parameters. A custom parameter — *0x92* has been added to the CPUID instruction. This parameter is passed in the *eax* register and takes the virtual address of the secure memory in the *ebx* register.

The secure application code containing the CPUID instruction is trapped in the VMM. The VMM recognizes the operand 0x92 and passes the virtual address to the SAPI component described in vBASE framework. In the SAPI component, the virtual address is translated into Guest Frame Number (GFN) and eventually into Machine Frame Number (MFN). This MFN is then made read-only and a Virtual Interrupt (VIRQ) is sent to the DOM 0. The VIRQ is handled by the kernel module in the DOM 0 and it notifies the Monitor through a signal. The monitor prints the confirmation message to the user and the secure process resumes its execution. Since the secure memory pages are protected by the VMM, even overwriting requests from OS will be ignored. The *xm daemon* will report if any such attempts to overwrite the read-only (secure) pages is done.

3.3.1. Source Code Implementation

In the listing 3.1, a chunk of memory (secure_mem) of PAGE_SIZE is allocated and the beginning of the chunk of the memory is aligned to beginning of a page.

```
char secure_mem[PAGE_SIZE] secure_memory __attribute__((__aligned__(
   PAGE_SIZE)));
```

Listing 3.1. Secure Memory Allocation

In listing 3.2 we declared a long pointer (secret_key) and type casted the char buffer pointer to an long pointer (secret_mem). The secret_key is assigned the address of the secure_memory. Now the contents of the address contained in secret_key will be stored in the chunk of memory allocated in listing 3.1.

```
long * secret_key=(long *) secret_mem;
*secret_key = 654321;
```

Listing 3.2. Secret Key Linked to Secure Memory

The address of the secret key is to be protected by making the memory location of the secret key read only. In listing 3.3, we pass the secret key to the hypervisor using ring3_hypercall. This is an assembly macro which takes the virtual address of the secure memory and a command integer as parameters. In the macro these parameters are passed as operands for the CPUID instruction. The CPUID instruction will then trap into VMM.

Along with the virtual address we send a command to the SAPI module in the hypervisor which decides the action to be performed on the virtual address. Initially we will send a PROTECT command. This will protect the virtual address by making the corresponding Machine Frame Number read-only. Originally Xen allows hypercalls to be done only through privileged interface in the operating system. But we have solved this problem by using CPUID instruction.

We call this mechanism as ring3_hypercall, which stands for hypercalls made from ring3 of DOM U.

```
int ring3_hypercall(unsigned long gva, int cmd)
{
        int ret;
   __asm__ __volatile__(
        "cpuid"
        : "=a"(ret)
        : "a"(0x92), "b"(gva), "c"(cmd)
        :"cc", "edx"
        );
        return ret;
}
unsigned long gva=(unsigned long) secret_key;
ret = ring3_hypercall(gva, PROTECT);
```

LISTING 3.3. Pass the virtual address to the VMM

Hypercalls are trapped into the VMM, thus giving VMM the control over process execution. The function of the VMM, where hypercall is handled is called as the hypercall handler. The Guest Virtual Address (GVA) is received in the hypercall handler and Machine Frame Number (MFN) is computed which is the hardware machine's RAM frame number. The function that computes MFN needs Guest Frame Number (GFN) as the argument. So first we computed GFN using paging_gva_to_gfn() function.

This GFN is then passed on to the gmfn_to_mfn where MFN is calculated. This MFN is marked read-only by the xen's in built function. This function requires previous type (read-only, write-able) of the MFN. Hence we first invoke a function which returns the old type of the MFN. The MFN is passed along with the old type and new type (p2m_ram_ro). This will update the MFN type in xen's tables. Any further requests to overwrite the pages identified by the MFN are dropped by xen. Attempts for trying to write to the read-only MFN are reported to DOM 0 through Xend. This process is shown in listing 3.4.

```
struct vcpu *v=current;
struct domain *d=v->domain;


p2m_type_t old_type;
mfn_t mfn;


gfn=paging_gva_to_gfn(current, gva, &pfec)


mfn=gfn_to_mfn(d, gfn, &old_type);
p2m_change_type(d, gfn, ot,p2m_ram_ro);
```

Listing 3.4. Translate GVA to MFN and mark MFN read-only

In listing 3.5 the code sends a global VIRQ to the DOM 0. In Xen, VIRQ's are handled by the guests as interrupts. So we need to bind the VIRQ to an IRQ Handler. After the VIRQ is populated from the hypercall handler, the DOM 0 is resumed and an interrupt (dynamically bound by the kernel) is issued. This will trap the control into the IRQ handler bound earlier.

```
send_guest_global_virq(dom0, VIRQ_VBASE);


bind_virq_to_irqhandler(VIRQ_VBASE,0,vbase_handler,NULL,NULL, 0);
```

Listing 3.5. Dispatch VIRQ from Xen

In listing 3.6 the task_structure of the Monitor process is found using find_task_by_pid function provided by the kernel (note that pid of the monitor can be sent through a system call or by writing to kernel filesystems). After finding the task_struct a signal is issued to the Monitor process. This will notify that the virtual address of the secure process has been made read-only.

```
find_task_by_pid_type(PIDTYPE_PID, monitor_pid);
send_sig_info(SIG_VBASE, &info, task_structure);
```

LISTING 3.6. Dispatch a SIGNAL to the Monitor

Therefore it is evident from the prototype implementation, that the implemented security architecture detects any unwanted modification to the secure application memory. These modifications can result from malicious applications, VMs, vulnerable OS and network channels. The secure architecture embedded in the vBASE framework, for implementation purposes, employs an encryption and integrity verification mechanism. This allows the security architecture to detect malicious modifications and eventually discard them.

Similarly, different hardware secure architectures could be embedded with vBASE to bolster the security features in cloud computing.

3.4. CTrust Framework[2]

We then extended vBASE architecture and implemented in a cloud environment. The motivation for the proposed *CTrust Framework* is to provide security to the applications running in a cloud by rooting there trust in the underlying VMM/hypervisor.

3.4.1. CTrust Architecture

The intuition behind the CTrust architecture is to deploy the vBASE framework in a cloud environment to provide security and root of trust to the applications running in the cloud. Figure 3.3 shows the proposed *CTrust* Architecture. This represents a private cloud built on top of the vBASE framework. In this architecture, the cloud is composed of a cluster of real computing machines known as the physical nodes. The nodes can be easily added to the cloud based on its load. Each node is connected to each other via a physical network. The *CTrust* architecture is implemented on XCP 1.0 [1] that contains XEN Hypervisor version 3.4.x and CentOS with 2.6.32.x Linux kernel. Since the vBASE framework is developed

[2]This section of CTrust has previously appeared in S. Nimgaonkar, S. Kotikela, M. Gomathisankaran, 2012 Cloud Computing Academy of Science and Engineering 152-165. Reproduced with permission from ASE Journal.

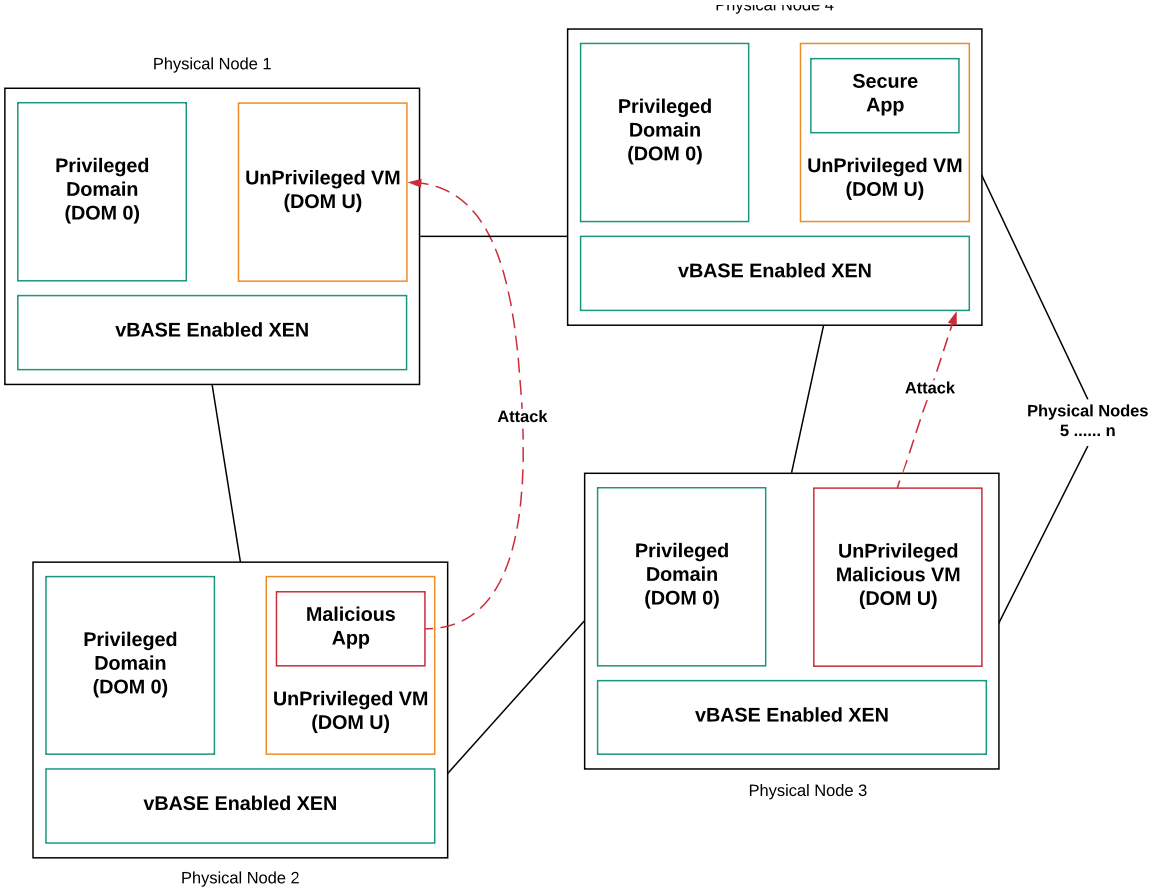

FIGURE 3.3. CTrust Cloud Framework

by modifying the XEN hypervisor, it is possible to replace the original XEN hypervisor in XCP 1.0 with the vBASE (modified hypervisor) framework. However it is important to emphasize that the mechanisms used to develop both vBASE and CTrust are generic to any virtualization software.

The CTrust architecture is implemented with one HVM unprivileged virtual machine. A HVM domain is a type of XEN VM which is capable of running unmodified operating systems (e.g. Windows). XEN originally did not support unmodified operating systems. However Intel added support for this in it's Intel VT technology [134]. Now it is possible to run unmodified operating systems in HVM domain. The HVM domain has been chosen to implement CTrust for two reasons. The first reason was to present a solution which doesn't require changes to operating systems. The second reason is that the HVM guests have great

paging support available in XEN API. However, it is important to stress that the same solution would be easier to port to PV domains also. The primary reason for this is that a PV domain runs a modified OS which is aware of the changes done by the underlying VMM.

3.4.2. Security Analysis

The root of trust for the security of CTrust is the vBASE framework. The vBASE framework keeps track of the execution of the protected cloud application by monitoring all the events in the system. Whenever the application's execution is interrupted, vBASE encrypts and hashes all the memory pages of the application. When the application resumes its execution vBASE checks for the integrity of the pages before allowing it to execute. Thus both the confidentiality and integrity of the application is protected by CTrust. The novelty of CTrust is that it not only provides protection mechanisms but also means to record events for auditing. The event trigger mechanism of the secure hypervisor report the events, application level, OS level, network level, and hardware level, to the Monitor program, which creates a log that can be used to prove that the application executed as expected.

3.5. Related Work

3.5.1. Hardware based secure execution

In the recent years, Intel SGX [41, 42], has become a potential alternative to the yesteryear hardware security architectures. Intel SGX feature is available as an extension, through additional instructions, to the general purpose CPU. Intel SGX provides confidentiality and integrity guarantees similar to vBASE. However, it inherits some of the challenges of the hardware security architectures. Being implementing in the hardware, Intel SGX may have unforeseen security issues [69] that can become hard to debug and fix. By leveraging the software layer, it is much more practical to design flexible security architectures that can be tested and improved as necessary.

3.5.2. Process-level Isolation using Virtualization

[24] *dfork* is a clone of posix fork in which a separate kernel and filesystem is allocated to every new process to isolate its operations and interactions of other processes with the

secured process. This method also gives an ability to review the changes done by the application before committing to the underlying hardware. Thus giving us the control to accept some changes, keep some isolated, and discard others entirely. dfork is also recursive, in the sense that a secured process can in turn spawn another secure process. Our architecture doesn't try to modify or override existing OS functionalities. It adds minimum overhead (hypercalls and VIRQs) to secure applications.

3.5.3. Quebes OS

[71] Joanna Rutkowska introduces a new (modified Linux) operating system called Quebes OS. It is very similar to dfork in a way that every process that is being secured gets its own kernel and environment in a virtualized container. The quebes approach uses a lightweight templating mechanism to create new virtual machine for every process to be secured on the fly. A copy of light-weight operating system (assumed to be secure) is saved in the control of VMM and whenever a user wants to run an application securely, a new virtual machine based on the secure copy is instantiated and the application is run in the virtual machine and only that application is run in that virtual machine. The (isolation) security provided by the Quebes OS is based on the (widely accepted) fact of strong isolation provided by Xen. Our architecture doesn't use any more resources than actually necessary for the application. Instead of running the application in its own virtual machine to isolate, our architecture focuses on security critical code and tries to isolate only security sensitive code in the VMM using existing VMM functionality.

3.5.4. Singularity

[65] Microsoft OS research proposes a novel approach to solve the process isolation problem by introducing a new concept of Software Assisted process isolation as opposed to widely accepted, de-facto, standard of hardware assisted process isolation. Singularity addresses the performance penalties of hardware based isolation and how this penalties force OS developers to break the intended design and make the OS insecure. Software assisted isolation tries to exploit the advancements in language technologies such as objects and messages and advocates usage of advanced languages to build secure operating systems. In our

architecture we don't have to introduce new operating system, environment or applications. We have shown that applications can be secured with in the existing environments without much effort.

3.5.5. Framework for Design Validation of Security Architectures

Framework for Design Validation of Security Architectures [48] showed how security architectures can be implemented in virtualized environments and explained how security components can be embedded in virtualization layer to mimic hardware security architectures in the virtualized environments. The framework is then shown to be used to test security architectures and rapid prototyping of security architectures. This was implemented using VMware is more of a testing framework.

3.6. Future Work

Security through virtualization technology is a very promising and upcoming field. There is lot of research going on exploring many ways to leverage the widely used virtualization. It would be no surprise if future computer systems ship with virtualization software along with firmware. Which means virtualized solutions for security would be more common place in coming future. We can build more functional security components in the virtualization layer. These components can be exported to applications through an API. This would help application developers in building secure applications with less effort and no rework.

The monitor can be made more powerful by adding more functionality like looking into the entire process memory and detect malicious activity while safe guarding the security intensive memory regions. Monitor can also be made interactive, where it receives a set of commands to monitor specific functions/memory regions, events through an input file. We can create an environment which automates various types of attacks on the secure process and let the monitor get the effects of the attack. This gives quantified data about the effectiveness of the security architecture. As many businesses are moving towards cloud computing [43] which is majorly deployed over virtualization layers, we can implement our

architecture in public and private clouds to ensure various challenges such as: confidentiality and data integrity [151]. Also, in future, when virtualization becomes prevalent on the desktop systems our architecture can be used to ensure the security of the applications.

3.7. Summary

We have successfully shown how security architectures can be built and implemented easily and efficiently using virtualization technology. We have proposed a Virtualization Based Secure Execution and Testing Framework and implemented its prototype using xen virtualization layer. The framework proves to be secure against unauthorized over-writing of security sensitive memory locations. By removing Operating System from the Trusted Computing Base and rooting the security of the application in the hypervisor, we can have stronger security guarantees.

CHAPTER 4

RACE-FREE ON-DEMAND INTEGRITY MEASUREMENT[1]

4.1. Introduction

In Chapter 3, we saw how we can provide secure execution in a virtualized environment. Along with the secure execution environment, it is important to have verified trust in that environment. In this chapter, we will review some of the existing solutions for establishing a trusted environment and how they fall short to provide dynamic and on-demand trust guarantees. Further, we will present our own solution Radium, to provide on-demand integrity measurements.

In the PC platform, a chain-of-trust is established from hardware to application by the hardware measuring firmware, firmware measuring the system software, and the system software measuring the application. This chain-of-trust is anchored in hardware which is the Core Root of Trust for Measurement (CRTM). CRTM is an immutable function, which measures the first component in the chain, and can be implicitly trusted. There are two approaches to establish a chain-of-trust in PC platform. The first approach, known as Static Root of Trust for Measurement (SRTM) [36], measures all the components sequentially during system boot and can ensure that the application is running in a trusted environment.

Conversely, the Dynamic Root of Trust for Measurement (DRTM) [66] can establish the chain-of-trust at any time of system execution. DRTM is invoked by special instructions to create an isolated and measured execution environment for the secure software. DRTM uses virtualization technology to create and protect this isolated execution environment. One of the typical usecases of DRTM is to instantiate the isolated execution environment during system boot (called here as DRTM-at-boot), by the boot loader. This is used to setup a trusted environment at the time of system boot.

The fundamental axiom, on which trust is built in SRTM and DRTM-at-boot ap-

[1]Majority of this chapter has previously appeared in S. Kotikela, T. Shah, M. Gomathisankaran, G. Taban (2015) International Conference on Privacy, Security, Risk and Trust (PASSAT). Reproduced with permission from ASE journal.

proaches, is that the system components do not change after they are measured. When this axiom is broken either through the vulnerabilities or through design flaws, a *race condition* is created between the time of measurement and time of use leading to TOCTTOU attacks.

One way to avoid this race condition is to use an approach followed by Flicker [92]: synchronize measure, launch, and use of the trusted application with the time of check. However, this approach limits the type of trusted applications because the overhead in setting up a trusted environment is very high. The alternate approach is to use synchronize the measurement and use of the trusted application which can be asynchronous to the launch. In other words, measure the application and environment whenever it needs to be used for security sensitive tasks irrespective of when it is launched.

Flicker [92] uses DRTM to invoke an isolated and measured execution environment for the application any time after boot. In DRTM only one isolated environment is allowed to execute. The system would initially boot normally into an untrusted environment from which, DRTM instructions are invoked to launch an isolated trusted execution environment. Subsequently a small Piece of Application Logic (PAL) is executed in this environment. This PAL is completely isolated from the untrusted environment including the runtime provided by the operating system. As there is no interaction with the untrusted environment, PALs are built with limited libraries and functionality. Minimal Trusted Computing Base (TCB) is also another motivation for limited PAL size which leads to limited functionality. This limits the type of trusted applications or services supported by PAL architecture. In this architecture the PALs or the trusted services are to be invoked by the untrusted environment. This approach gives the untrusted environment the ability to deny the launching of the trusted service or modifying the state of the untrusted environment to spoof the trusted service. For example a root-kit can unload itself before invoking a trusted root-kit detector service.

To address these problems, we propose Radium - Race-free on-demand integrity measurement architecture. Radium uses a control domain, which interfaces with the hypervisor to launch measuring services. The Radium architecture uses this measuring service for on-

demand integrity measurement. The measuring service is is invoked whenever the software component needs to be used and thus prevents TOCTTOU attacks. Radium architecture provides an access control mechanism to allow trusted services access to untrusted services and communicate with other trusted services. This access control mechanism allows for fine granular access to untrusted components and enables measuring services to perform more semantic measurements than just cryptographic hash.

In this chapter we present a prototype of Radium architecture designed using Xen hypervisor. We use a DomU Virtual Machine (VM) as our measuring service. Enabling us to measure other virtual machines running on top of Xen. We also present an example application, Trusted Rootkit Detector. We have noticed that using a measuring service is more efficient than measuring the entire untrusted environment.

4.2. Related Work

In this section we will study some of the projects that relate to Radium. With the advancement of TPM and Trusted Computing enabled hardware many projects have created various systems that support trusted computing environment.

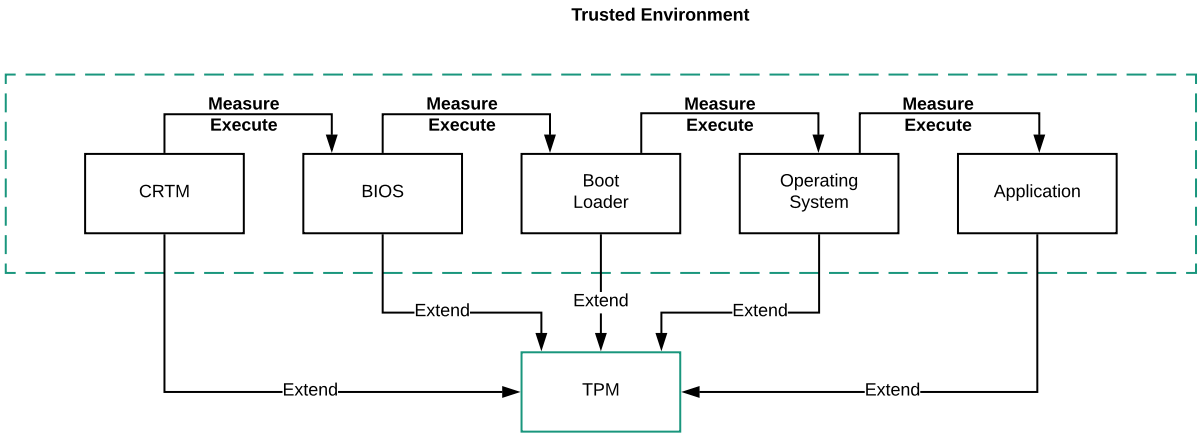

FIGURE 4.1. Static Root of Trust for Measurement

4.2.1. SRTM and DRTM

SRTM and DRTM are two trusted computing solutions that use TPM to set up a measured trusted environment for a trusted application to execute.

In SRTM approach, as shown in Figure 4.1, hardware CRTM measures the platform configurations and the BIOS, BIOS measures the bootloader, bootloader measures the OS kernel, and OS kernel measures the application to be launched in a trusted environment. After each component is measured, the measurement is extended into the Platform Configuration Registers (PCR) of the TPM. The TPM version 2 has 24 PCRs. SRTM uses the static PCRs (0–15) to store the measurements and these static PCRs are reset at the time of system boot. These measurements together with platform configurations form the state of the platform. In SRTM it is ensured that each measurement is done by a trusted (previously measured) component. As the SRTM chain-of-trust includes bootloader, SRTM can only be established during the system boot. Hence every time a trusted measured environment is to be instantiated using SRTM, a system boot has to be performed.

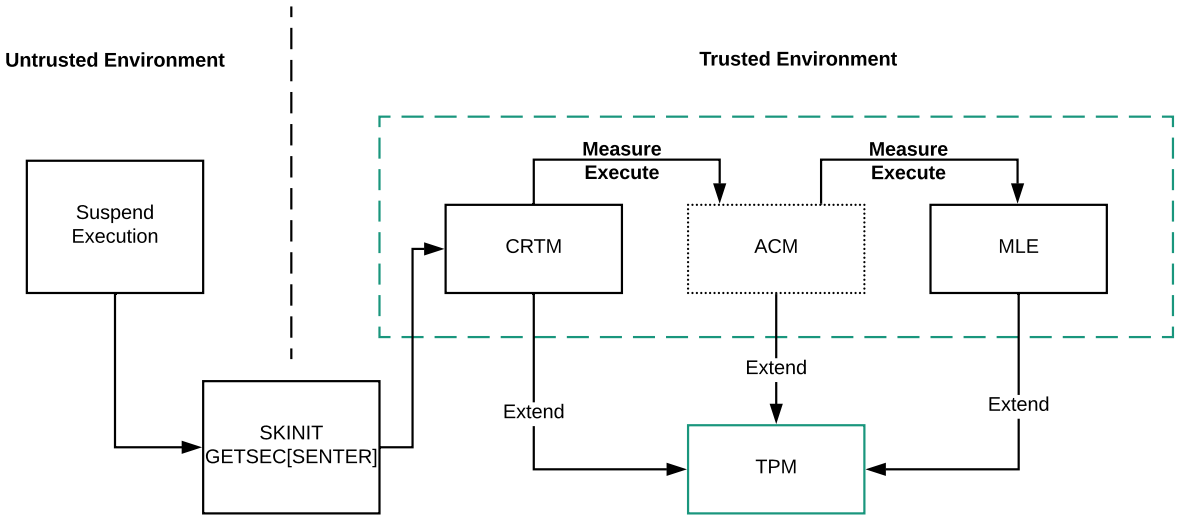

FIGURE 4.2. Dynamic Root of Trust for Measurement

In DRTM approach the chain-of-trust is initiated by special platform specific instruction. The DRTM invoking instruction takes a memory address as a parameter. This memory address points to the program that needs to be measured and executed in an isolated environment, commonly known as Measured Launch Environment (MLE). A DRTM isolated execution environment is set up as follows. Once the DRTM setup instruction is invoked,

the processor suspends all threads of execution (if any) and starts a new isolated thread of execution, as shown in Figure 4.2. The CRTM measures an optional Authenticated Code Module (ACM) and the measurement is extended to the TPM's PCR. In turn the ACM measures the MLE and the measurement value is extended into the TPM. After extending the state of the MLE into TPM, virtualization is enabled and the MLE is loaded into the memory with virtualization protection. The virtualization protection restricts access to the loaded program from any other process or firmware, thus keeping it secure from any possible attacks. Then the MLE is given control to execute.

The first measurement of the DRTM chain-of-trust is extended to the dynamic PCR 17. This PCR can only be extended by the hardware CRTM and this extension is performed only with the special DRTM instruction. As the special DRTM instruction suspends all other threads of execution, possibly malicious, it eliminates all threats of attack and ensures the security of the program running in DRTM. DRTM can be invoked at anytime including at the system boot. DRTM at the boot time is similar to SRTM in functionality except that, DRTM is more secure with virtualization protection enabled. Typically DRTM is used by the boot loader to launch measured hypervisor program during the system boot.

4.2.2. AMD SVM and Intel TXT

AMD Secure Virtual Machine (SVM) [22] and Intel Trusted eXecution Technology (TXT) [67] are two platform solutions which provide SRTM and DRTM support. SVM uses special instruction called `SKINIT` to invoke a DRTM environment and TXT uses `GETSEC[SENTER]` instruction to start the DRTM environment. Both these instructions take a memory address as the parameter. This memory address points to the program (hypervisor or an application) that should be executed in an isolated execution environment.

Intel TXT first verifies the digital signature of ACM and validates it. The corresponding key for the ACM's digital signature is hardcoded into the platform. The ACM's validation is extended to PCR 17 and the validated ACM is executed to ensure that the platform has the necessary support for DRTM. Then the ACM measures and extends the measurement of the program (pointed by the memory address parameter) into the PCR 18.

In case of AMD SVM, there is no ACM. So, the chain-of-trust starts with the measurement of the program pointed by the memory address. Once validated, virtualization protection is enabled and the program is loaded into virtualization isolated memory space. Once validated the program is given control over to execute. After this the program has access to all the dynamic PCRs. So the TPM quote executed by the program can contain dynamic PCRs and proves that the program has been measured by the trusted hardware.

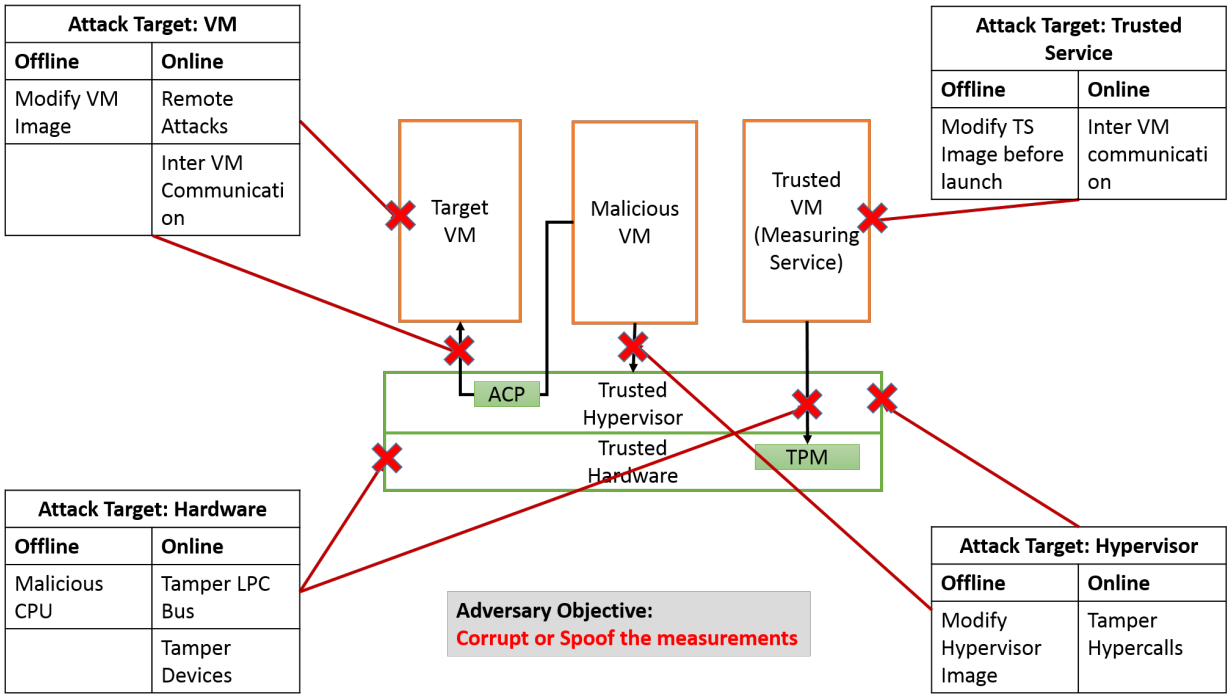

FIGURE 4.3. Adversary Model

4.2.3. Measured Boot

Trusted GRUB [133] is one of the first trusted computing solutions which used SRTM to provide the measured boot. `tboot` [38] is a project from Intel that uses DRTM to provide measured boot. By using DRTM, tboot enables memory isolation from other environments as well as any malicious I/O devices. The Open Security Loader (Oslo) [108] is an open source boot loader focusing on security through minimal TCB. The entire Oslo codebase is around 1000 lines and with such minimal code base, Oslo can be more thoroughly tested for bugs and vulnerabilities. In [33] Butterworth et al., analyzed CRTM and presented

certain vulnerabilities in the CRTM. Time sensitive attestation has been proposed to fix problems with these vulnerabilities. All these solutions use TPM to store and report the measurements.

4.2.4. Trusted Hypervisor

Terra [53] is a trusted hypervisor solution which uses cryptographic functions to securely isolate the virtual machines. Terra supports two types of VMs, namely closed-box VM and open-box VM. The closed-box VM is used to implement proprietary hardware platforms such as gaming hardware. All the code and data of a closed-box VM is encrypted and protected even from the host owner. Open-box VMs are used to run open systems such as personal computer. Each VM is strongly isolated from other VMs and there is access for closed-box VM to a open-box VM or vice-versa. TrustVisor [91] is a hypervisor as root-of-trust solution and uses software TPM for measurements. TrustVisor is a single rich guest only hypervisor. It lets one single rich guest execute on top of it and this single rich guest can invoke security sensitive Piece of Application Logic (PAL) dynamically. When the PAL is executing, all the untrusted OS and applications are suspended. PALs are self contained code executing independently. Due to lack of the common application development libraries, PALs are built with very limited functionality. Since TrustVisor design is modeled after DRTM, the PAL executes in complete isolation. The PALs don't run concurrently neither there is any inter-PAL communication.

4.2.5. Hypervisor based access control

Xen and KVM are premier opensource hypervisor solutions. Both these hypervisors support Mandatory Access Control (MAC) [116, 31] allowing a fine granular policy enforcement. They filter the hypercalls from the virtual machines and allow only those hypercalls that are conforming to the policy. These MAC mechanisms are modeled after SeLinux [90]. Both systems provide a secure sandbox environment for multi-tenant virtual machines on a single host.

4.2.6. Rootkit Detection Solutions

The virtualization based solution presented in [157] is geared towards designing a rootkit detector for guest virtual machines via deep information extraction and reconstruction at the hypervisor level. The virtual machine memory is reconstructed by locating static data entries of the kernel system map. The guest VM will extract its own memory and the hypervisor will extract the guest VM's memory from the shadow pages. A comparison will be performed against the two memory snapshots. Whether the extracted states are identical or not cross-verification will also be performed. A socket communication will be used between the hypervisor and the untrusted guest VM and the detection can be initiated by either side. Taking memory snapshot of the whole VM and comparing it to the OS report is a time consuming process compared to measuring only necessary pages. The Copilot [113] is designed to detect malicious modification to a host kernel on a separate PCI add-in card with a low performance cost to the host machine. Depending on the configuration the Copilot monitors and examines the host RAM to detect malicious malware through Direct Memory Access without the knowledge or intervention of the host kernel. Since it runs on its own PCI it does not rely on the correctness of the host and is resistant to tampering from the host. It periodically computes hashes over key parts of memory that impact kernel execution, compares against a known value, and reports to an external system that enables manual decisions to be made regarding detected changes. Copilot adds the cost of the hardware and this hardware cannot be programmed to implement other trusted services in the hardware.

4.2.7. Hypervisor based monitoring and introspection

Due to various advantages of using hypervisor as the root-of-trust, many monitoring and introspection security solutions have been proposed [27] using Virtual Machine Introspection (VMI). By using HyperAgents [32], agents that run on behalf of the hypervisor, many security services can be provided to the guest Operating Systems. This also reduces the necessity of running such a solution inside every guest. In [131], Tang et al., demonstrates how to provide an AntiVirus solution using the VMI. By utilizing the VMI, it is possible to not have any in-guest agent. This can improve detection rate against malware that changes

its behavior in presence of monitoring agent. These monitoring and introspection capabilities can also be extended to Cloud Computing systems with minimal overhead [145].

4.3. Adversary Model

In our adversary model, the objective of the adversary is to spoof the measurements stored in the TPM. We classify the adversary based on the attack target and the time of attack. Attack targets are various components of the architecture the adversary can attack to achieve this objective. Adversary can attack these components *offline* — before the components are launched or used, and *online* — while the component is being used. Figure 4.3 shows various attack targets an attacker would be aiming attacking and possible exploits that can be performed.

The various attack targets we consider in our adversary model are described in the sections below.

4.3.1. Hardware

At hardware level, the adversary can replace or modify the CPU with a malicious CPU offline. This malicious CPU is capable of spoofing presumed good measurements to the TPM irrespective of the state of the software running. This will allow attacker to run malicious software without being detected. While the system is online, the attacker can attach malicious devices that can steal information and gain access to various information passing through the system interconnections and also the LPC bus. These attacks will leak sensitive information from the system including various TPM commands and measurements.

4.3.2. Hypervisor

When the system is offline, an attacker can attempt to modify the hypervisor binary and make it malicious. As hypervisor runs at the highest privilege level, a malicious/-compromised hypervisor can give attacker control over the measuring service and all other environments running on the hypervisor. This will let attacker feed incorrect hash values to the measuring service and spoof a known good state of the system. While the system is online, an attacker can run a virtual machine or a trusted measuring service on top of

the hypervisor. Attacker can use these interfaces to attack the hypervisor via hypercalls. Vulnerabilities in the hypercall API can lead to privileged access of the hypervisor to the attacker.

4.3.3. Target VM

Target VM in Radium is typically used to run applications and services. Radium integrity measurement aims to ensure the trustworthiness of the VMs. While the target VM is offline, attacker can try to manipulate the VM image and corrupt it. While it is online, attacker can use other VMs to launch an inter-VM channel attacks or use remote attacks via network.

4.3.4. Trusted Service

Trusted Service is a crucial component in Radium's architecture. As the Trusted Services are implemented using a VM, the same attacks applicable for the VM are applicable for the Trusted Service. By compromising the Trusted Service, an attacker can spoof the state of the target VM, in which user applications and services are running, and present it to be in a known good state.

4.4. Radium: Architecture Overview

In this chapter we propose a novel Race-free On-demand Integrity Measurement Architecture (Radium) that uses DRTM and Virtualization mechanisms to provide a measurement architecture that allows on-demand measurements to be performed as and when needed. On-demand measurements are independent of booting the system and launching the environment. This means, an application can be launched any time and used any time while ensuring the trust through measurement at the time of using the application. Radium consists of a hypervisor, access control policy, one or more measuring services and any combination of trusted and untrusted environments. The root-of-trust for Radium is provided by Asynchronous Root of Trust for Measurement (ARTM).

We use an example scenario to explain the architecture. Consider a platform which has a VM running on a hypervisor. This VM is required to execute a security sensitive

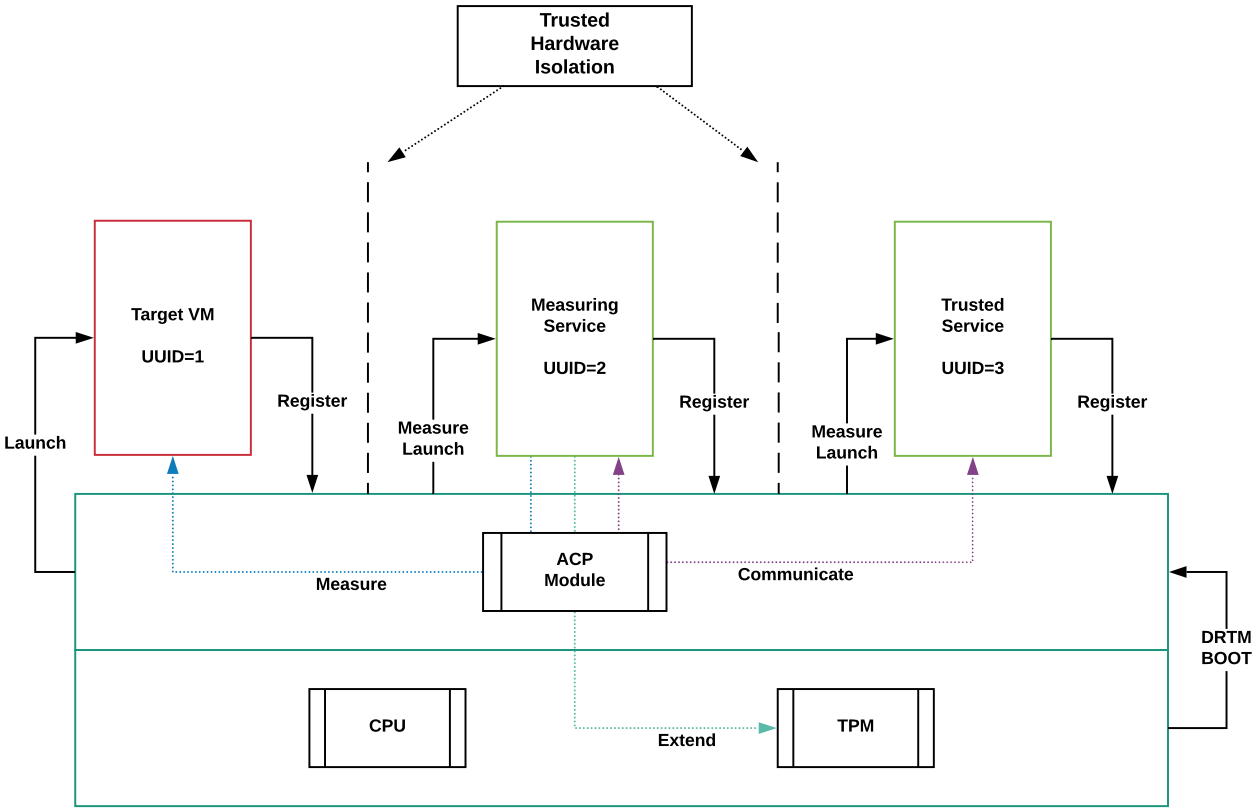

FIGURE 4.4. Radium Architecture

application in a trustworthy manner and prove its execution to a third party verifier. In DRTM such a scenario requires that the VM be launched when the application needs to be executed. Whereas Radium allows on-demand measurements and the VM can be launched and measured independently. Thus in Radium, before executing the security sensitive application it is possible to measure VM and check if it is trustworthy at the time of use. Figure 4.4 shows various components of Radium architecture and communication between these components. Each individual component is described in the following paragraphs.

4.4.1. ARTM

We propose a novel Asynchronous Root of Trust for Measurement (ARTM) architecture which uses both Virtualization Technology (VT) and DRTM that enables asynchronous use of an MLE. We achieve this by creating a Measuring Service to provide *use-time* mea-

surement services to other MLEs. In ARTM the hypervisor is brought up using DRTM at system boot. After the hypervisor is launched by the hardware DRTM, any target VM can be launched. A measured execution environment that provides measurement services to other environments will be launched to verify the trustworthiness of the target VM on-demand. This execution environment/VM is known as Measuring Service (MS) in Radium architecture.

The key difference between DRTM and ARTM is that the isolation in ARTM does not require other execution environments to be suspended. The hypervisor in Radium is trusted to provide strong hardware isolation between all the processes running on top of it, even when these processes are running concurrently. As Radium relies on the hypervisor to provide trusted isolation between all virtual machines and the measuring services, the hypervisor becomes root-of-trust for isolation and measurement of trusted services. This reliance on hypervisors for root-of-trust is justified as hypervisors have many orders small size compared to the operating systems making them more trustworthy than the operating systems.

Using hypervisor as the root-of-trust is a common practice in contemporary security research [53, 91, 140]. This usage of hypervisor as the root for security solutions has encouraged the research efforts in minimizing the TCB of hypervisor [139, 125], making them more modular [102] and minimizing the runtime interface of the hypersvisor resulting in shrinking [74, 57] the attack surface. Automated formal verification of hypervisor has been studied by Alkassar, Hillebrand, Paul, & Petrova [21]. Thus a formally verified hypervisor with minimal TCB and minimal runtime interface can extend the security and trust provided by hardware mechanisms such as DRTM to the applications.

4.4.2. Hypervisor

Hypervisor is a system software capable of running operating systems as processes. In Radium, the hypervisor is a type-1 bare metal hypervisor which can run directly on top of the hardware. Type-1 hypervisors typically have many orders smaller codebase and consequentially, many orders smaller TCB. The hypervisor in Radium architecture is capable

of running unmodified operating systems as well as applications like Unikernels [86]. This hypervisor uses hardware mechanisms to isolate all VMs running on top of it. The hypervisor exports hypercall (analogous to system calls in an operating system) for the applications and services to interact with it. As the services in Radium architecture need to communicate among each other, the hypervisor provides inter-VM communication mechanisms.

In Radium, hypervisor along with the trusted hardware mechanisms form the Trusted Computing Base (TCB) and is trusted to perform as expected. To ensure the trustworthiness of the hypervisor, it is always brought up using DRTM. The hypervisor includes an Access Control Policy (ACP) module which is used to control interactions between various VMs and applications. This ACP can use any access control mechanism such as Mandatory Access Control (MAC) or Capability Based Access Control. All VMs are managed through an Application Programming Interface (API) exposed by the hypervisor. This API allows local as well as remote users launch VMs and measuring services as needed.

4.4.3. Trusted and Untrusted Environments

In Radium we define an environment as trusted if is measured, verified, and launched. A trusted environment can provide any specific trusted service if the environment is used immediately after the launch thus preventing any TOCTTOU attacks. An environment that is launched with no measurement is considered to be an untrusted environment. These two types of environments can co-exist and execute concurrently along with the trusted services. Trust can be extended to an untrusted environment by a trusted measuring service by measuring and verifying the untrusted environment at the time of its use.

Each environment in Radium is uniquely identified by an Universal Unique IDentifier (UUID). This UUID is assigned to each environment during its creation and is used for registration with the hypervisor. This UUID is used by the hypervisor internally to identify, allocate resources to, and control each environment individually.

4.4.4. Measuring Service

Measuring service is a trusted environment that measures other environments (target VM). The measuring service is measured, verified, and launched only when another environment (VM) is to be measured. This service is used immediately as soon as it is launched and is shut down as soon as it is done measuring. This immediate usage of the measuring service ensures that it is not subjected to the TOCTTOU attacks. Radium requires at-least one measuring service to provide on-demand measurement and it can support more than one measuring service for different types of measurements.

Different measuring services may differ in the type of measurements they perform, for example one service may measure the kernel for root-kit and other measuring service may measure the environment for malware. Measuring services need to have necessary accesses and permissions to perform the measurement. These permissions are granted through the access policy rules. In a typical virtualized environment any virtual machine with necessary permissions can act as a measuring service. Radium allows different measuring services to communicate and cooperate with each other.

In Radium, hypervisor uses TPM to store the measurements of trusted services and measuring services. The measuring services also use TPM to store the measurements of target VMs. As the hypervisor is launched with DRTM, PCR 17 contains the measurement of the hypervisor (18 in case of Intel TXT as the SINIT ACM's measurement is stored in PCR 17)and the remaining dynamic PCRs 18/19—22 are in the control of the hypervisor. The hypervisor uses these PCRs to save the measurement of various measuring services and trusted services. The measuring service uses PCR 23 to save the measurement of a target VM.

4.4.5. Access Control Policy Module

Radium allows access between its environments through the Access Control Policy Module (ACPM). ACPM is part of the hypervisor and executes in privilege level of the hypervisor. The ACPM contains an access policy for the entire architecture. The access policy is composed of rules that identifies the accessor environment and its corresponding

accesses. ACPM monitors all accesses between the environments and allows only the accesses that comply with the policy. Access requests that do not comply with the policy are denied.

If no policy rules are provided for an environment, it is subjected to a restrictive default policy of no access. A non-trustworthy environment is never given access to the resources of a trusted environment. Even the trusted services get only necessary access permissions to the untrusted environment, for example to check the presence of a root-kit in an untrusted environment the measuring service needs only read permissions to the target environment. This prevents any malicious programs acting as trusted service to attack a target environment.

Whenever any environment tries to interact with other environment, ACPM uses UUID to identify each accessor and the accessee. After identifying the accessor, accessee, and the operation accessor is trying to perform ACPM verifies from the policy if the accessor has necessary permission. If the accessor has necessary permission, the operation is permitted, else it is blocked by the ACPM.

4.4.6. Working

In this section we describe the working of various components of the Radium architecture.

Trusted Hypervisor Setup The hypervisor in Radium architecture is the first component to be launched. It is launched during the system boot using DRTM. The boot-loader uses special instructions to invoke a DRTM MLE and prepares an isolated execution environment as explained in Section 4.2. Then the hypervisor is measured and compared with a previously known good value which is stored in TPM non-volatile RAM (NVRAM) [38]. Depending on the platform the hypervisor is running on, AMD or Intel, the measurements are stored in the PCR 17 or 18 respectively. If the measurements are same as the previously known good value, the hypervisor is deemed trustworthy and executed. The hypervisor then takes control of the platform.

Measuring Service Registration Measuring Service needs to be registered with the hypervisor before it is available for measuring other environments. During registration pro-

cess, the Measuring Service provides all the ACPM rules necessary for its operation to the hypervisor. Hypervisor adds these rules to the ACPM policy and links it with the UUID of the Measuring service. Also, the hypervisor measures the Measuring Service and saves the known good value of the Measuring Service for future comparisons. The Measuring Service is encrypted and the key is *sealed* into the TPM. At this point the state of the platform contains the measurements of known good value of hypervisor. Thus the key is unsealed only if the platform is running a known good value hypervisor.

Target VM Registration and Instantiation

Target VM is a normal VM that runs the hypervisor. The target VM also needs to be registered with the hypervisor before executing. During this registration set of policy rules are provided to the hypervisor which contain UUID of the Measuring Service that can access this particular target VM and with what permissions. Hypervisor registers the UUID of the Target VM and the Target VM will be instantiated after its registration.

Launching the Measuring Service Measuring Service is invoked to measure the Target VM via hypervisor's Application Programming Interface (API) calls. A Measuring Service has to be invoked while the Target VM is running, else appropriate error is reported. Before launching the Measuring Service, the image needs to be decrypted. The necessary key is only unsealed if the platform is executing a known good value hypervisor. In the presence of the known good value hypervisor, the key is unsealed and the Measuring Service image is decrypted. Then the Measuring Service is measured and compared with its previously known good value. If the measurements match, the Measuring service is launched. Else the launch is denied. The Measuring Service's measurement is extended to PCR 19.

Measuring and Attesting of the Target VM Measuring Service uses hypercalls to access the Target VM. When a Measuring Service requests access to the Target VM state, the ACPM verifies if the Measuring Service has necessary permissions or not. If the Measuring Service has necessary permission, the Measuring Service is allowed to access the necessary memory pages and perform appropriate measurement. The Measuring Service will then report the measurement of the Target VM to the hypervisor through the toolstack API. The

hypervisor extends the measurement to the TPM's PCR 23.

To attest the target VM, the verifier would require the complete state of the system. The measurements of the platform and hypervisor will ensure that the system is indeed running in a trusted environment and thus the target VM's measurement is trustworthy. The verifier will request a quote from the TPM. The TPM will send a quote containing all the necessary PCR values and this quote is signed by TPM's Attestation Identity Key (AIK). The verifier will verify that the quote has indeed come from the appropriate TPM via it's AIK signature and verifies the Target VM's measurement.

4.5. Prototype Implementation

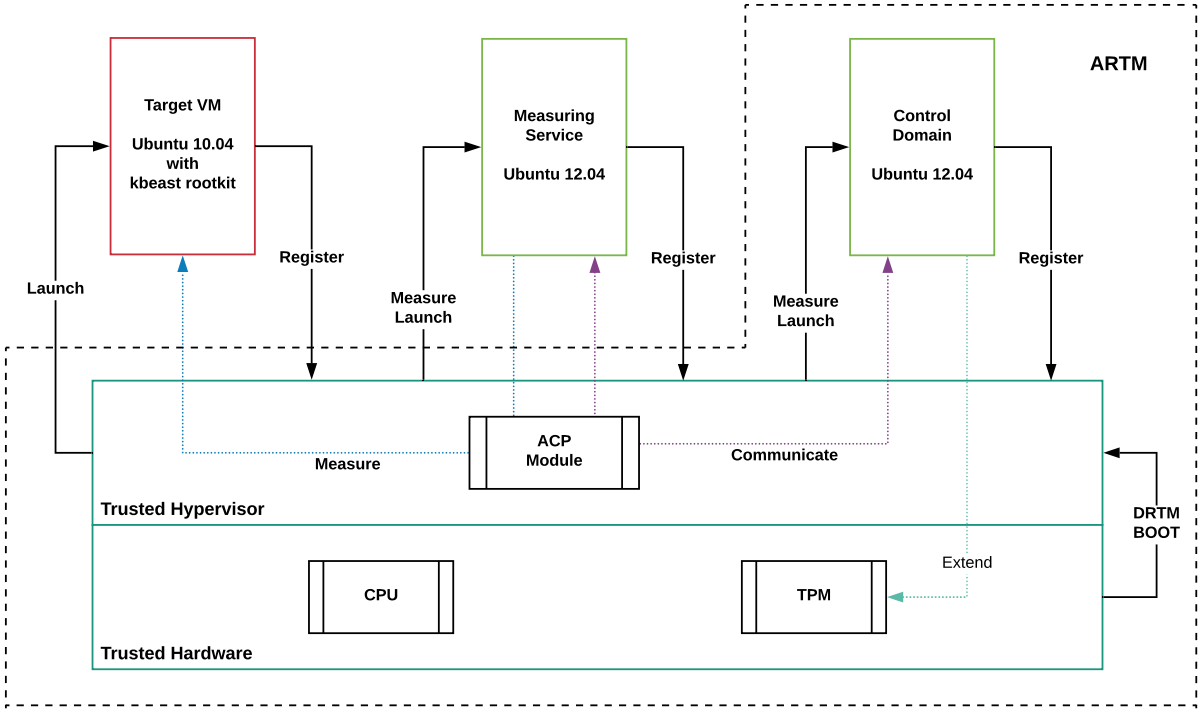

FIGURE 4.5. Radium Prototype

In this section we describe the details of the prototype, on which we implemented a trusted root-kit detector, that uses Radium. None of the available commodity hypervisors

have all the necessary features ideal for Radium architecture. To demonstrate the capabilities of our architecture, we have built a limited prototype using Xen hypervisor.

Xen hypervisor has all the necessary functional requirements. However it has larger TCB and not-so-minimal hypercall interface. The modular architecture of Xen is more suitable than other type-1 bare metal hypervisors for the Radium architecture. This modular architecture of Xen separates all the management responsibilities of the hypervisor to a special control domain called Dom0. The Dom0 provides necessary drivers to interact with the hardware on behalf of the Xen hypervisor. Dom0 is essentially an extension of the hypervisor.

As shown in the Figure 4.5, the ARTM in the prototype consists of trusted hardware, trusted hypervisor and the control domain (Dom0). As ARTM is asynchronous the untrusted environment can be used anytime for a security critical application. But whenever the untrusted environment is to be used, the measuring service is invoked and the it measures the untrusted environment. If the measurements match a known state, the untrusted environment is trusted for usage.

In our prototype, we used Ubuntu 12.04 LTS for the Dom0 and another Ubuntu 12.04 LTS for Measuring Service DomU. The Target VM is built using Ubuntu 10.04 and is infected with kbeast [73] rootkit. The hardware used is an 4th Generation Intel i5 processor with VT-x, VT-d and Intel TXT (Intel's DRTM implementation) enabled. The working of the prototype is discussed in the following paragraphs.

4.5.1. Boot-up

The Xen hypervisor is booted with DRTM. This ensures that we are booting the hypervisor into a known trusted state. After BIOS has finished POST and the control is transferred to the boot loader, the boot loader (tboot [38] in our case) invokes the special DRTM instructions, `GETSEC[SENTER]` with Xen's kernel address as parameter. This will measure and load the Intel's SINIT ACM only if the SINIT ACM's digital signature is verified. The SINIT ACM will verify the TXT hardware and starts preparing the environment for measured launch. The Xen kernel is measured and the hash is compared to a known

value. If the Xen is proven to be unmodified, the VT-d protections are enabled to create an isolated environment. With these protections in place, the Xen kernel is loaded into the memory and the control is given to the Xen.

4.5.2. Protecting Confidentiality of Measuring Service

To ensure the confidentiality of the measuring service we encrypted it using a secret key created in the Dom0. This encryption key is sealed in the TPM with the known hypervisor state, i.e., the known good state of the Xen. The measurements for this DRTM environment are stored in the PCRs 17 and 18. Hence we use these PCRs in our sealing operation shown in the following command.

```
tpm_sealdata -z -i secret.key -o ./secret.blob -p17 -p18
```

4.5.3. Launching a Measuring Service

Whenever we want to launch the Measuring Service, we have to unseal the key and decrypt the measuring service. The key will be unsealed only if the platform is in the known good state with correct values of PCR 17 and 18. This is done using the unsealing operation shown in the following command.

```
tpm_unsealdata -i ./secret.blob -o secret.key -z
```

If the platform is not in a known good state, the secret key will not be unsealed and the trusted launch will be aborted. If the platform is in a known good state, the Measuring Service is measured and launched. This measurement is stored in PCR 19.

All the measurements in this prototype use the physical TPM, which means that only one measuring service can execute at any point of time. Alternatively virtualized TPM [144] which allows each measuring service to have its own set of PCRs in a vTPM, can be used. This will allow different measuring services to measure different target VMs concurrently.

4.5.4. Measuring the Target VM:

Radium allows feature rich measurement of applications with greater semantic knowledge of the measured environments as compared to the DRTM environments. In our prototype, we used root-kit detection as an example. The measured environment has been

infected with `kbeast` rootkit. `kbeast` is a stealthy rootkit which uses system call hooks and hides its presence. Even though it has a module inserted in the kernel and a Trojan process running, these are never reported by the operating system. We detect this module and Trojan process' presence using the trusted root-kit detector (measuring service).

The memory of all the DomUs are isolated in separate address spaces by Xen. In order for the measuring service to introspect the target VM, measuring service needs to have access to the memory pages of the untrusted environment. To enable this, we provide necessary policy rules. Xen has an inbuilt Mandatory Access Control mechanism known as Xen Security Modules (XSM). XSM is an integral part of Xen and administers all interactions between the VMs (environments). Each VM is addressed by a security label in XSM. This label is setup through the policy configuration file. To enable the measuring service access the memory pages of untrusted environment, we need to write necessary policy rules. The XSM policy rules allow us to write rules with hypercall level granularity. We can define what hypercalls each environment can execute. With necessary labels and policy rules the measuring service will be capable of measuring the target VM.

The measuring service will invoke a hypercall asking a copy of the necessary memory pages of the untrusted environment. The ACPM will then check if the policy permits the measuring service to access the requested memory pages. If the policy permits, the hypervisor will give access to necessary pages to the measuring service. The measuring service will read all the necessary pages. In the case of rootkit detector, the measuring service accessed all the necessary kernel pages. Then we use `volatility` [142] to walk through these kernel pages to detect the hidden module and process. The following command is used to list hidden modules in the untrusted world and happens in real time.

```
vol.py -l vmi://TargetVM --profile=ubuntu linux_check_modules
```

The volatility accesses the `libVMI` link to the target VM's memory and the profile (type of Operating System), executes the given command. We have used `linux_check_modules` to list all the hidden modules in the kernel at the moment the analysis is being performed.

4.5.5. Attesting the Verification:

After the root-kit detector introspects the untrusted environment, the result will be saved to the TPM. The measuring service will handover the measurement to the hypervisor to write it in the TPM. In our prototype example, we used PCR 23 for this. The PCRs 17, 18, 19 and 23 form the final state of the platform after root-kit detection. A remote party can verify the result of the root-kit detection by requesting a quote. The TPM can send a quote containing valued of the PCRs 17, 18, 19 and 23. With these values the verifier can perform trusted verification if the root-kit detector has run in a known good state inside a measured environment and what is the result of the root-kit detection.

4.6. Performance Analysis

In traditional trusted computing solutions we either reboot or reset the entire the system to bring up a MLE. We have identified performance critical phases in these solutions and we aligned these performance critical phases of Radium and measured the times taken for each of these phases.

The four performance critical phases in traditional trusted computing solutions are 1. Measurement time of the Target VM 2. Launch time of the Target VM (happens during boot time or reset) 3. PCR Extend operation and 4. Quote operation. In Radium architecture the measurement of Target VM is performed by the Measuring Service and the Target VM will not be reset or reboot. However to guarantee that the measuring service is an unmodified known-good-version, we launch the Measuring Service every time we want to measure the Target VM. Hence the launch and boot times of the Target VM in traditional trusted computing solution is replaced by the launch time and boot time of the measuring service.

The performance critical phases in Radium are 1. Time taken to measure the Measuring Service VM 2. Launch time of the Measuring Service VM 3. Measurement time of the Target VM 4. PCR Extend operation and 5. Quote operation. Out of these the TPM operations, `extend` and `quote` are present in both the solutions and these times are not incurred by Radium itself.

We have noticed that it takes an average of 26.7 seconds for measuring Target VM with SHA1 hash and an average of 35.8 seconds for booting the Target VM. Making it an average of 62.5 seconds to ensure trustworthiness of the Target VM in traditional trusted computing solutions. In Radium we measured, on an average, 27.3 seconds to measure the measuring service, 11.1 seconds to boot the measuring service and 1.7 seconds to measure the Target VM. Making it a total of 40.1 seconds. These measurements are dependent on the size of the Target VM image, Measuring service image and number of services running in the Target VM. Services in Measuring Service VM are minimal and it is possible to make a single application virtual machine images with no unnecessary services as shown in [86].

Two key take aways from this analysis are: In the traditional trusted computing solution the Target VM has to be suspended for around 62.5 seconds and reset to the initial state. There is no down-time for the Target VM in Radium, as the Measuring Service can perform live measurements in real-time. Also, the traditional TC solutions have to measure the entire Target VM image for verifying its trustworthy state, which takes around 27 seconds. Whereas in Radium, the Measuring Service reads only necessary memory pages, which for rootkit detection, has taken less than 2 seconds.

4.7. Security Analysis

Radium architecture derives security from the hardware mechanisms. Radium uses TPM and DRTM features to provide trust, isolation, and security for its components. In this section we will describe how Radium provides security for each of its components. In the Table 4.1 we compare how various Trusted Computing (TC) solutions compare with Radium in providing security to various components.

4.7.1. Hardware

Any attack at the hardware level is difficult to detect and prevent. Various offline attacks on the hardware described in the Section 4.3 will not be detected by Radium or any other TC solution. These attacks are transparent to and happen outside Radium's control. When the system is offline, Radium is not executing and it has no way to know that the

TABLE 4.1. Comparison table of various trusted computing solutions

Attack Target	Hardware		Hypervisor		Target VM		Trusted Service	
TC Solution	Offline	Online	Offline	Online	Offline	Online	Offline	Online
SRTM	No	Yes	Yes	No	Yes	No	No	No
DRTM	No	Yes	Yes	No	Yes	No	No	No
Other TC Solutions (Flicker, TrustVisor etc.)	No	Yes	Yes	No	No	No	No	Yes
Radium	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes

attack is happening and hence cannot detect or protect itself from the attacks. Thus we include the CPU, TPM and the Hypervisor in our Trusted Computing Base (TCB) and assume they are not altered. While the system is online, an attacker may attach certain devices to the hardware and leak information. However leaking this information alone will not alter the system behavior. Even the leaked TPM measurements will be of little help to an attacker. The DRTM measurements are initiated by hardware with a single thread of execution. Any tampering to the software state will be detected by the Radium.

4.7.2. Hypervisor

Hypervisor is the most privileged layer of software in Radium's architecture controlling all other components. An attacker can attack the binary of the hypervisor when the system is offline. This attack will be detected when the hypervisor is launched at the system boot and the DRTM measurements of the hypervsior will change with a changed binary. The online attacks on the hypervisor comes through various interfaces the hypervisor exposes for other components to interact with it, like hypercalls and API calls. An attacker may run a target VM or trusted service on the hypervisor and use it to attack the hypervisor. The minimal hypercall interface in the Radium's hypervisor will decrease the attack surface. The ACPM will limit what hypercalls a target VM or trusted service can invoke and thus prevents from malicious VMs or services attacking the hypervisor. The trusted hypervisor with minimal TCB, minimal hypercall API, and a ACPM module will withstand the online attacks.

4.7.3. Target VM

Target VM is any VM that is not trusted and typically used to run user applications. These applications may sometimes need to perform security sensitive operations and deal with security sensitive data. An attacker would try to attack the target VM image while it is offline and change the application binary so that it may leak sensitive data. Radium uses Trusted measuring service to measure the target VM state right before it is being considered for a security sensitive operation. the measuring service would assess the state of the target VM and can detect any alteration or manipulation to its state. While the target VM is online, the attacker can use other VMs or trusted services to launch attacks via inter-VM communication. Radium uses ACPM to prevent any such attacks as it allows only permitted VMs talk to the target VMs through the inter-VM channel.

4.7.4. Trusted Service

A trusted service is a VM used to provide trusted services like measuring service. Being a VM it is subject to the same attacks as the target VM. When, offline the VM image can be attacked. But trusted service VM images are protected using encryption. And the encryption key is sealed with the TPM. The key will be unlocked only when the system (the hardware and the hypervisor) is in a known good state. All trusted services are invoked via measured launch. A trusted service will only be launched if the measurement of the trusted service matches a known good value. If not, the launch will be aborted. When the trusted service is online, it is protected by the Radium's ACPM policy.

4.8. Summary

As users and enterprises entrust more and more sensitive data and functionality to computing platforms, from handheld devices to servers, the trustworthiness of the system becomes a central problem. Numerous academic studies have presented how trusted systems and trusted execution environments can be created and used. However, existing solutions suffer from the TOCTTOU race condition. They often sacrifice performance and/or functionality of the trusted and measured environments. In this chapter we proposed a novel

Race-free On-demand Integrity Measurement Architecture (Radium) that addresses these significant limitations. Radium allows feature rich on-demand measurement of applications with greater semantic knowledge of the measured environments as compared to the SRT-M/DRTM architecture. We designed a trusted root-kit detector using the Radium and demonstrated its usefulness.

CHAPTER 5

ONTOLOGY BASED VULNERABILITY MANAGEMENT FOR CLOUD COMPUTING[1]

5.1. Introduction

In Chapter 4, we have studied how we can provide on-demand trust guaranteed for applications running in the virtualized and cloud environments. To keep up with the vulnerabilities in the software stack, we need to have robust vulnerability management program. In this chapter, we will present a novel technique to manage vulnerabilites using Ontologies and then demonstrate how the Ontology Vulnerability database can be instrumental for managing vulnerabilities in a cloud computing scenario.

Security vulnerabilities are prevalent across all facets of software. The vulnerabilities are increasing every year at an exponential rate. Our experience with software engineering shows it is very difficult, even impossible, to build software without vulnerabilities, because of the complexity of modern software systems. So the only way to deal with vulnerabilities is find them and patch them. Discovering and patching vulnerabilities is not an easy task. To deal with this complex vulnerability management, we need standard and efficient methods and tools.

The first step to deal with vulnerabilities is classifying them. Vulnerability classification is a well-studied area in computer security. Many vulnerability classifications have been proposed and devised. Most of them have chosen the taxonomy [30] approach to classify vulnerabilities; however, many of these classifications have proven to be inefficient, incomplete or erroneous. In taxonomy based classification the elements being classified are divided into groups and sub-groups (hierarchy). Hence the taxonomy approach requires assigning vulnerabilities to one and only one sub-group. But many times vulnerability would be present in more than one sub group. This can be due to incomplete and/or incorrect definition of

[1]Majority of this chapter has been previously published from S. Kotikela, K. Kavi, M. Gomathisankaran, The 2012 International Conference on Security & Management (SAM 2012) 67-73. Reproduced with permission from CSREA Press

the vulnerability or the subgroup. It has been observed that this situation arises due to the nature of vulnerabilities themselves [96] [59].

Vulnerabilities are concepts, not entities themselves. It is natural for them to overlap across different groups. Ontologies are better suited than taxonomies to model concepts. Ontology [155] is a knowledge representation technique which is used to model real-world concepts and their relationships [37]. It is one of the prominent techniques used to model and share a domain specific knowledge in the field of information science. Ontologies are widely used in artificial intelligence, semantic web, and library science where classification of concepts is very essential. These properties of ontologies make them perfect candidate for vulnerability classification. A rich collection of existing tools and frameworks will make creating ontology based vulnerability classification easy and efficient. The structured nature of ontologies makes it easy to reason, query and infer. These features of ontologies have led to adoption of ontologies in many security solutions such as [137] [61] [18] [112].

As Cloud Computing [43] continues to expand and evolve, it is influencing the way we think about computing. Every aspect of computing is now connected to cloud computing. It is a big game changer across all verticals of computing. This demands a lot of attention and research for cloud computing. The Cloud Security Alliance had mentioned that security is one of the biggest roadblocks in adopting cloud computing. As many businesses and users are adopting and using cloud, there will be lot of software running in the cloud. Vulnerability management is still relatively new. This makes the problem even more interesting [151] with respect to cloud computing.

In this chapter we present a solution for vulnerability management in cloud environments. Our solution uses well defined ontologies. The proposed framework consists of Ontological Vulnerability Database, Semantic Natural Language Processor and Attack Code Database. We designed an ontology by extending the Ontology for Vulnerability Management (OVM). Then we designed a framework around the ontology and created an Ontological Vulnerability Database (OVDB) which has semantic collection of vulnerabilities. The OVDB is linked to an attack script database in which there is a many-to-many mapping between

vulnerabilities of the OVDB and scripts of the attack script database. The attack script database is a collection of attack scripts which will invoke runnable attack code from the attack code database. The attack code database is compilation of attack codes from popular attack codebase like Metasploit. This attack database can be used to launch attacks on applications to test for the associated vulnerability. A natural language processor will facilitate natural language and keyword search on the OVDB. The semantic nature of the ontologies will facilitate the reasoning and inferences on the OVDB. The framework facilitates vulnerability scanning and vulnerability assessment of an application. This work can be further expanded to assess the runtime environment by extending the ontology to include configurations of the environment.

The rest of the chapter is organized as follow. Section 5.2 outlays some background concepts related to our work. Section 5.3 describes the Related Work. Section 5.5.3 explains the various Ontologies. Section 5.4 explains the architecture of our proposed framework. Section 5.5 presents the Implementation of our framework followed by the Future Work in Section 5.9 and Summary in Section 5.10.

5.2. Background

Common Vulnerabilities and Exposures (CVE): CVE [3] is a publicly available listing of vulnerabilities and exposures in software systems. This project is initiated and maintained by MITRE organization. CVE doesn't attempt to classify the vulnerabilities. It just enumerates all the vulnerabilities. Every vulnerability in CVE has a unique identifier, description and list of software systems along with corresponding versions that are affected by this vulnerability. This public repository helps many other vulnerability research projects. The CVE project started by the MITRE organization now lies at the core of many security/vulnerability research projects. Our framework also depends directly on CVE repository at its lowest level; however, there are many refined layers available on top of CVE, such as NVD. We will be using them than the raw CVE format.

Ontologies are at the heart of our research and many vulnerability assessment projects. In this section we will provide brief introduction about ontologies. Ontology is defined as "A

formal explicit description of concepts in a domain of discourse, properties of each concept describing various features and attributes of the concept, and restrictions on properties. Ontology is a conceptualization of a domain of interest". It consists of concepts, relationships between these concepts and rules specifying the limitations of these relationships. The concepts from the real world are modelled as classes in ontology. The members of these classes can be individuals (real-world-objects) or other classes or a combination of both. The properties model various attributes of the individuals or the properties of the classes in general. Properties are also used to model relationships between two individuals or classes.

Ontologies are expressed in ontology languages. OWL [143] is the World Wide Web Consortium (W3C) standard for representing ontology. OWL stands for Web Ontology Language. There are 3 sublanguages for OWL: OWL-Lite, OWL-DL and OWL-Full. The three languages differ in their expressiveness.

OWL-Lite is the simplest of the three. It is used where simple hierarchy and simple constraints are sufficient. It is easy to build ontology in OWL-lite and it is the best choice to migrate an existing taxonomy/hierarchy to an ontology using OWL-Lite.

OWL-DL is based on Description Logics (DL) and is more expressive than the OWL-Lite. The inclusion of DL in OWL can be exploited for automated reasoning due to the First Order Logic properties. It is also the most used OWL variant by many researchers.

OWL-Full is required for situations where high expressiveness is desired. It is to be chosen when high expressiveness is more essential than decidability or computational completeness of the language. OWL-Full cannot be used for automated reasoning.

5.3. Related Work

Security Content Automation Protocol (SCAP): SCAP [9] is a suite of interoperable specifications for automating security management. SCAP is a standard developed by NIST along with community participation. By using SCAP protocol to build a security solution will ensure that a security solution will be interoperable with other related security solutions. Our proposed OVDB framework is SCAP compliant. SCAP is essential for bringing automation, standardization, and regularity to many security related initiatives. It is the de-

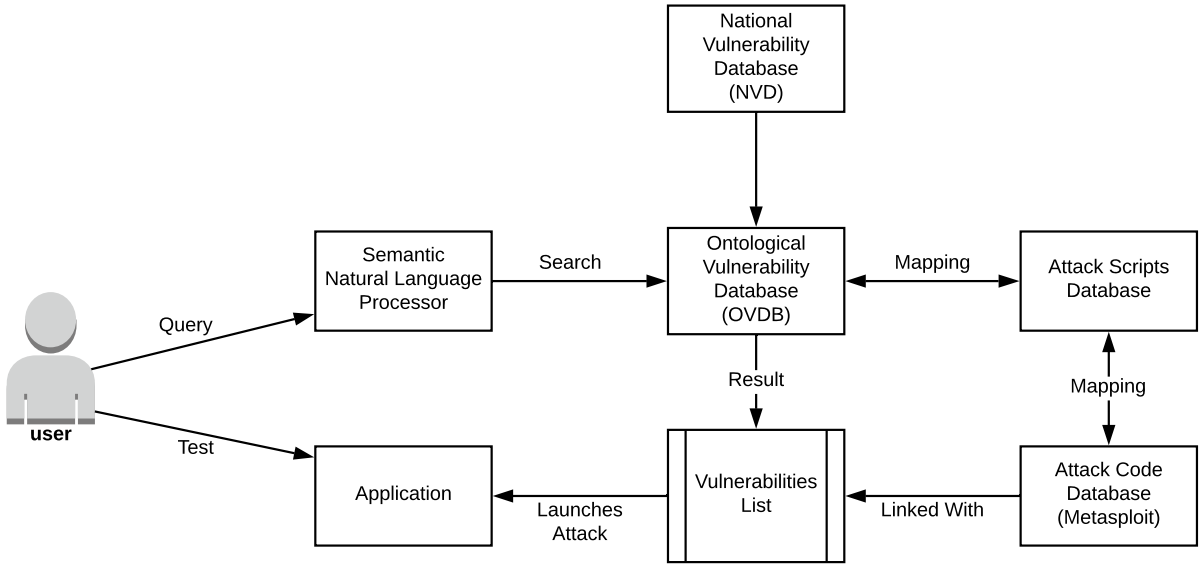

FIGURE 5.1. Vulnerability Assessment Framework

facto standard for achieving inter-operability between various security automation projects. Hence we also align our ontology with SCAP so that we can leverage existing fine works that are SCAP compliant and also ensure that our framework interoperable with other similar initiatives and projects.

National Vulnerability Database (NVD): NVD [7] is a SCAP compliant vulnerability database maintained by NIST. It is essentially the SCAP compliant version of the CVE enumeration. The NVD database is released as NVD feeds in XML format. This NVD database is used as an input for the OVDB creation. NVD is a refined version of CVE. It has all the CVE data and in a SCAP compliant format, hence using NVD makes security solutions more robust and more interoperable. In the same light we also use the NVD database as a source for our Ontological Vulnerability Database.

Ontology for Vulnerability Management (OVM): OVM [147] is ontology for managing vulnerabilities. Ontologies are more suitable to model vulnerabilities than taxonomies [96]. OVM uses ontology to store, classify and refer to vulnerabilities mapped from the NVD vulnerabilities list. OVM can be used to store and retrieve vulnerabilities. It can be queried using SWRL (Semantic Web Rule Language) through which we can perform semantic comparisons between two related products. OVM is the precursor for OVDB. Though

both the databases share many similarities, OVDB is modified to consist only concrete and dis-ambiguous components and is intended to apply for cloud computing use-cases also.

OVM Software Assessment Tool (OSAT): OSAT [148] is an ontology based software assessment tool. It is built on top of OVM. It uses all the vulnerability information present in OVM and tries to measure the security of software applications. It uses the CVSS scores of each vulnerability present in NVD and computes total security measure for particular software using a formula which sums up the Common Vulnerability Scoring System (CVSS) scores of all the vulnerabilities present in that software. OSAT is really useful tool and one of the first of its kind. It quantifies vulnerability assessment. Our Vulnerability Assessment framework is also similar to OSAT and has some interesting improvements like more user-friendly search.

Ontology Of Cybersecurity Operational Information: Is an ontology [130] developed for identifying cybersecurity information in cloud computing. The basis of the ontology is derived by applying cybersecurity operations that are prevalent in regular non cloud computing environments and applying them to cloud computing. The set of operations identified are generalized out of the cybersecurity operations performed by various cybersecurity practitioners in USA, Japan and Korea. Ontology of Cybersecurity Operational Information is Cloud agnostic and aims at assessing the cloud environment for vulnerability assessment. In future OVDB is going to combine the OVDB (which targets application vulnerability) with the Ontology of Cybersecurity Operational Information to create a complete end-to-end cloud vulnerability assessment.

Cyber Security Knowledge Graphs By leveraging the Cyber Security knowledge bases, it is possible to construct knowledge graphs [70]. Knowledge graphs are essentially knowledge bases focused on pragmatic implementation, querying, and correlation as opposed to capturing and exchanging domain knowledge. Utilizing the semantic interconnections, huge networks of data modeled as graphs, can be connected and queried from various perspectives [106] to provide reactive and proactive cyber-resilience. Furthermore, knowledge graphs can provide real-time insights into the highly connected data utilizing auto inferencing [149].

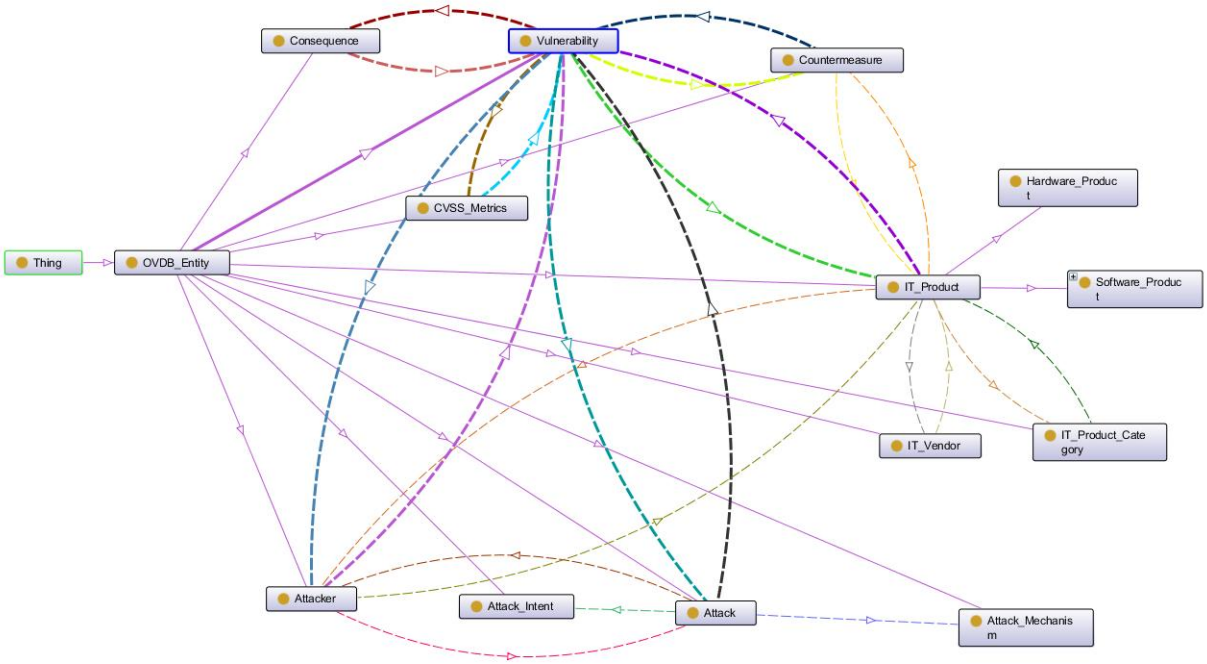

FIGURE 5.2. OVDB Ontology

5.4. Vulnerability Analysis Framework

In this section we will present major components of our framework followed by an algorithm.

Our Vulnerability Analysis Framework consists of *Semantic Natural Language Processor* (SNLP), *Ontological Vulnerability Database* (OVDB), *Attack Scripts Database*, and *Attack Code Database*.

Semantic Natural Language Processor (SNLP): The semantic capabilities of OWL ontology aids in performing semantic reasoning on the ontological vulnerability database (OVDB). The user enters generic or specific information about his application and the SNLP is responsible to search through the OVDB and pull out the vulnerabilities that match user's keywords. Certain keywords by the user can be used to reason semantically than just perform a keyword search/match. The SNLP is capable of performing both keyword search as well as semantic search.

Ontological Vulnerability Database (OVDB): OVDB is ontology database of vulnerabilities listed in the National Vulnerability Database. The OVDB includes lot of additional information about vulnerabilities like consequences, countermeasures, attacks that reveal a particular vulnerability etc. The ontology for OVDB is a modified version of the ontology found in OVM. There is a one-to-one mapping between OVDB and Attack Scripts database.

Attack Scripts Database: Attack Scripts Database is a collection of scripts which can invoke attacks from the attack code database. The scripts are customized for each attack individually as the parameters required can vary greatly for each attack. The scripts are mapped and a link to the script is stored along with associated vulnerability in the OVDB.

Attack Code Database: Attack Code Database is a database of attack codes primarily taken from Metasploit. The scripts in the attacks scripts database invoke the code in this database. This code will receive the parameters from the attack script and launches attacks on the application.

Algorithm 2 Working algorithm of OVDB framework

1: User enters keywords for the search
2: SNLP processes the user query and displays list of related vulnerabilities
3: User selects the vulnerabilities he wants to test the application for
4: User launches associated attacks for the vulnerabilities selected
5: Attacks are performed on user's application
6: A summary of attack results is posted for the user

The usage of the framework has been explained in Algorithm 2. The vulnerability assessment starts by user typing in the keywords which describe the application that is to be tested. This user query is submitted to the SNLP module. SNLP dissects the query and fetches various vulnerabilities related to the given keywords. These vulnerabilities will have a unique identifier (CVE-ID), brief description, impact score and a check box and launch attack button. After the user selected all the vulnerabilities he want to test, he can click Launch Attack button. This will invoke the associated attack script(s) from the attack-script database. The attack scripts will invoke necessary attack code from the attack code database.

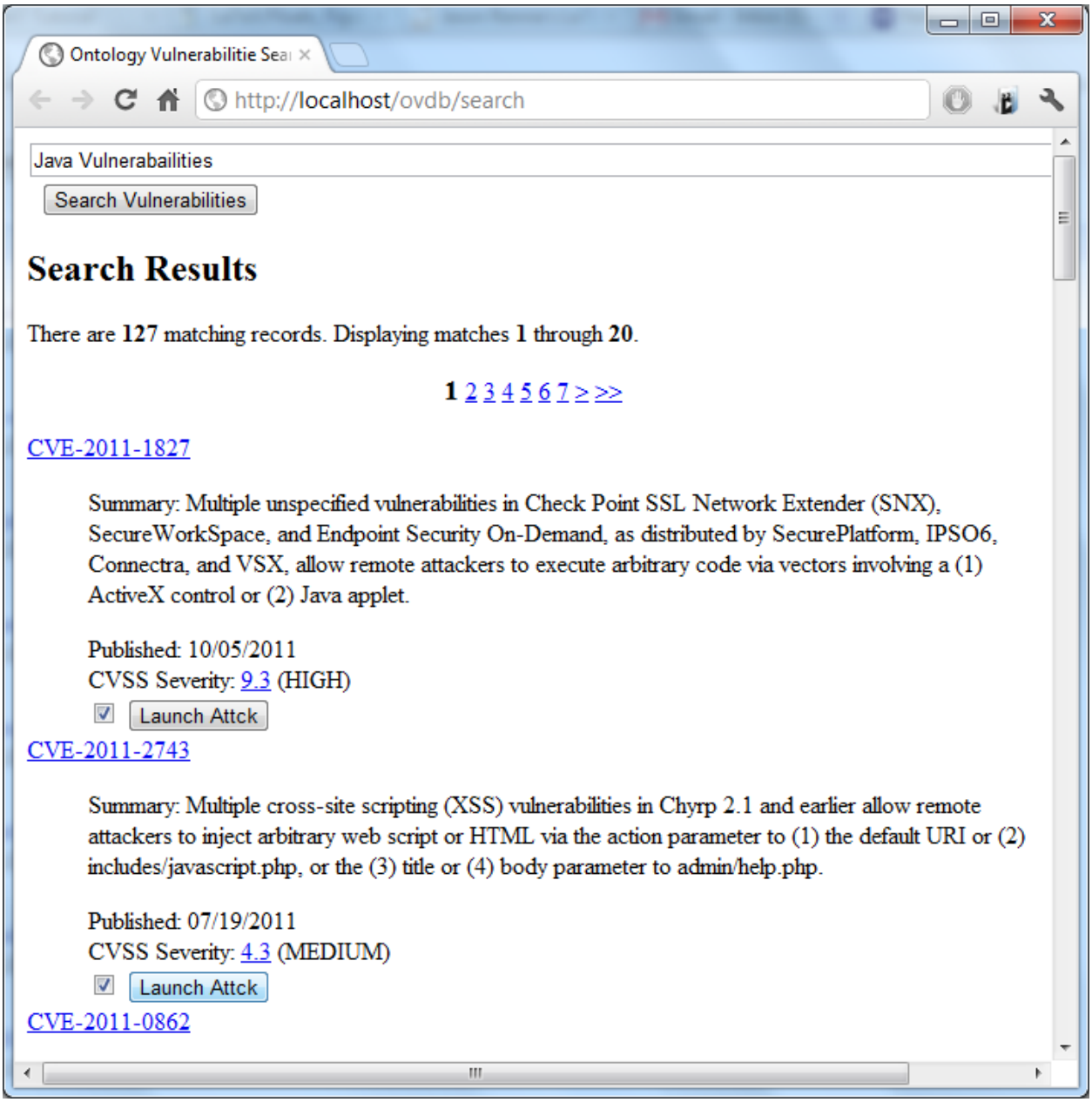

FIGURE 5.3. OVDB Framework Search Page

After all the selected vulnerabilities are tested, the user is presented with an analysis of what vulnerabilities have been tested positive and what have been tested negative.

5.5. Framework Implementation

The framework we propose is built on top of the existing state-of-art vulnerability assessment solutions such as OVM and OSAT and extends them with subtle modifications. Hence, to understand our framework, one will need a good understanding of OVM and OSAT.

5.5.1. OVM and OSAT

OVM is a vulnerability database (populated using NVD) which has a query interface. OVM can be queried using standard query language SWRL [107]. These queries go through the OVM and pulls out vulnerability information. An user can write queries and infer results very efficiently with SWRL. For example, if a user is looking for vulnerabilities in browsers, he doesn't have to perform his search for each browser individually. Instead, he can issue search terms querying to look up for vulnerabilities associated with applications like Firefox. The reasoner will automatically infer which applications in the database fall in the category (browser) as Firefox and pulls out all the vulnerabilities in those applications. SWRL is a robust and expressive language which allows users to perform customized and efficient queries according to their needs.

OSAT is a Security Assessment tool built on top of OVM. OSAT takes advantage of all the ontological properties of OVM and reports comprehensive and qualitative measurements of security. As the OVM, OSAT also follows the SCAP protocol. OSAT populates its reports from the OVM data. With OSAT we can enter a software product and ask the tool to list it's vulnerabilities. Alternatively we can provide input such as type of vulnerability, scope of the effect and nature of the vulnerability etc. Depending on user's input the OSAT infers the OVM database and builds a report of the vulnerability information. We can also ask the OSAT to find similar software along with security scores. This feature will allow us to compare which software product is more secure in a given product line.

5.5.2. OVDB Framework

Both OVM and OSAT are pioneering projects which have shown the power of using ontologies for vulnerability management. We build our OVDB vulnerability assessment framework as an extension to the ideas of OVM and OSAT. OVM and OSAT have reasoning and reporting which is limited to the applications in the database. They cannot be used for user created applications. OVM and OSAT reports details from the database depending on the user query.

These two tools report the vulnerabilities listed in the database. They cannot analyze the application and tell us what vulnerabilities are present in the application right now. This makes these tools static in nature, where we can only look up existing information. The OVDB framework aims at solving this problem. The framework includes an attack code database which is mapped to the vulnerabilities in the OVDB. Whenever an user wants to analyze his application, he will use the framework to search for vulnerabilities. The search module will report possible vulnerabilities. User can select and launch attacks to test corresponding vulnerabilities. This is explained in more detail below.

5.5.3. Ontology

We have developed an ontology for implementing OVDB. It is a modified and extended version of OVM. Figure 5.2 shows the various entities and their relationships between them. We have developed this ontology in Protégé(ontology editor) [12] . All the concepts in the ontology are derived from Thing (a generic entity signifying every child entity is thing). At the top we have a wrapper entity for our ontology, called the OVDB_Entity which signifies that every child entity belongs to OVDB framework. Vulnerability is at the centre of the ontology. It has relations with other entities like IT_Product, Countermeasure, Consequence, CVSS_Metrics, Attack and Attacker. The relationship of the Vulnerability with these entities is described below:

Consequence signifies that every vulnerability has a consequence. Having this information associated with the vulnerability helps us to search vulnerabilities using their consequence. Many normal users may not technically classify a vulnerability, but they can

identify the consequence and use it for searching the vulnerabilities.

Countermeasure entity contains the countermeasure for a vulnerability. It gives information on how to patch the associated vulnerability. This information will help users to patch their software and get rid of the vulnerability.

CVSS_Metrics is the set of CVSS metrics for a particular vulnerability. Which is a standard measurement for the severity of the vulnerability. It also tells which of the security properties of the information (confidentiality, integrity, availability) is being effected by the vulnerability.

IT_Product is the class of IT Products which have a particular vulnerability. This relation helps us to find vulnerabilities not only within the application but also the complete application stack. For e.g. if we are testing a Java Enterprise Application running in an application server, we can give the details of the application server to get the list of possible vulnerabilities in the application stack.

Attacker is the entity which is interested in exploiting the associated vulnerability. Having information about attacker will help to protect the application more efficiently.

Attack is the type of the attacks that can exploit a particular vulnerability. This allows user the flexibility to search if a particular attack is possible on his application. This relation will allow a quick evaluation of the application against dangerous attacks.

5.5.4. Working

Figure 5.3 shows a sample search results page. The user performs a search query by giving keywords describing the application such as technology, framework, language etc. (Java Vulnerabilities in the above example). The SNLP searches for the keywords in the OVDB and reports a list of vulnerabilities that are matching the user's keywords. User's keywords will be used for semantic search. After the search is done, the SNLP presents user with a list of vulnerabilities. These results are pulled out of OVDB. There is a check box after every vulnerability. The user can either choose some or all of the vulnerabilities and launch attacks corresponding to these vulnerabilities. User can click on launch attack button for every attack he wants to be performed upon the application. After the attacks

are performed on the application a detailed report is generated on the security status of the application.

5.6. Vulcan

We have further implemented our ontology model to perform vulnerability assessment in cloud computing context (VULCAN[2]). VULCAN uses ontology for creating vulnerability database and associates a vulnerability with one or more attack code snippets from the attack database. For assessing vulnerabilities in a specific domain, like cloud computing or mobile, we have organized our vulnerabilities into classes. In the following subsections the various components of VULCAN Architecture are described in detail and followed by working flow.

5.6.1. NVD

National Vulnerability Database (NVD) [7] is a SCAP [9] compliant vulnerability database. The NVD database collects vulnerability information from various interrelated vulnerability databases like CVE [3], CWE [5], CPE [2], CVSS [4] etc. and compiles the information into a single database. Every entry in the NVD database is identified by a unique identifier. This identifier is referred to as CVE ID, which is an unique identifier for each vulnerability in the CVE database. This is the same identifier used across various other vulnerability databases mentioned above. A typical vulnerability entry in the NVD database has the vulnerability identifier, description of the vulnerability, list of software and their versions in which this vulnerability is found in, vulnerability severity score (CVSS) etc. collected from appropriate vulnerability databases. These vulnerability databases are industry standard databases maintained by MITRE. All the vulnerability information found in these databases is contributed by volunteers across the industry. The SCAP compliance of the NVD database makes it easy to inter-operate with other security tools and automate security assessment. VULCAN uses NVD as the source to populate vulnerability information into the ontology knowledge base.

[2]The Vulcan section will also appear in the co-designer, Patrick Kamongi's (UNT) dissertation, "Ontology Based Security Threat Assessment and Mitigation for Cloud Systems - December 2018"

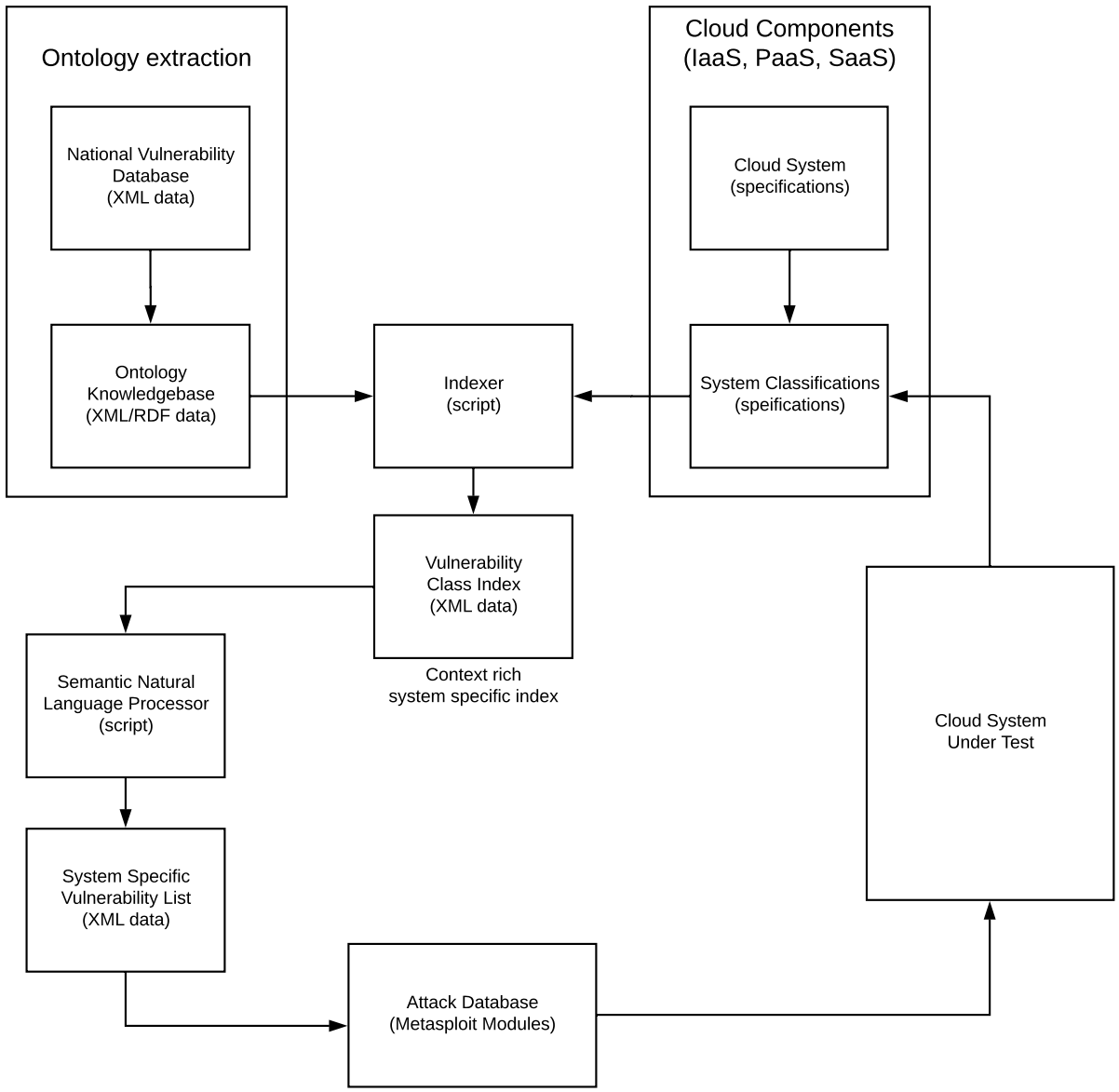

FIGURE 5.4. Vulcan Architecture

5.6.2. OKB

Ontology Knowledge Base is the ontological database of vulnerability information from the NVD database. NVD provides the vulnerability database in a XML feed. We extract the vulnerability information from the XML feed and populate ontology knowledge base. The vulnerability information in the NVD XML feed is present in various tags. All

the information in these tags are mapped to various classes and properties defined in the ontology.

5.6.3. System Classifiers

System Classifiers are dynamic inputs provided to the Indexer which will classify the classes in the ontology knowledge base. An example classification includes various vendors in the cloud computing domain and various software or hardware components in each service level of cloud computing services. As shown in Fig. 5.4, cloud computing domain is classified into IaaS, PaaS, SaaS etc. sub domains. In each of these domains we will include software and hardware components used in popular cloud computing vendors like Xen hypervisor in IaaS sub-domain, Google App Engine in PaaS sub-domain and Salesforce CRM in SaaS sub-domain. We can provide the system classifiers to whatever detail and depth we want to. The indexer takes these system classifiers as input and crawls through the ontology knowledge base and creates an index. The index consists of vulnerabilities grouped according the system classifiers provided by us. The changes in software or hardware in any domain or vendor would require updating the system classifiers and re-indexing the ontology knowledge base.

5.6.4. Indexer

Indexer is the software responsible for crawling through the ontology knowledge base an create an index. This index will in turn be used by the SNLP module to search the ontology knowledge base depending on the user query. The indexer is set to run every time the ontology knowledge base and/or system classifiers change. The indexer identifies all the vulnerabilities that are related to software or hardware components listed in the system classifiers and group them accordingly in the index.

5.6.5. Vulnerability Class Index

Vulnerability Class Index is the list of all vulnerabilities grouped into the categories provided by the system classifiers. These groups are called as "Vulnerability Classes". Vulnerability classes will assist users to search for vulnerabilities within a specific domain or sub-domain An example index looks like shown in the Fig. 5.4. At the top level there is

cloud computing class. Cloud computing has a sub class called PaaS and the PaaS class has Xen hypervisor as it's sub class. In the Xen class we have list of vulnerabilities extracted by the indexer from the ontology knowledge base.

5.6.6. SNLP

Semantic Natural Language Processor enables users to search and reason about vulnerabilities. It includes various sub components which are capable of doing pattern matching, keyword search, and reason over properties and relationships of the classes in the ontology knowledge base. SNLP takes input from user and tries to understand what the user is asking for and provides him a list of vulnerabilities for the requested product and/or class. SNLP is capable of looking up vulnerabilities for the requested product and listing vulnerabilities in a particular class or product across various vendors. It also can reason and list vulnerabilities for the technology or framework used in the user's application.

5.7. Vulcan Working Flow

The NVD database consists of vulnerabilities identified by CVE ID and available as a XML data-feed from NIST. The instances for each vulnerability will be populated using XML parsing techniques. The entire XML data-feed is transformed into Ontology Knowledge Base (OKB). The OKB has classes, properties for vulnerabilities and relationships between these classes. This knowledge base with classes (and respective instances), relationships and properties will enable us to perform semantic queries and reason about vulnerabilities.

After populating the OKB, we provide a dynamic set of classifiers to the indexer. These classifiers are used to classify the vulnerabilities in the OKB. The indexer groups various vulnerabilities into classes of a specific sub-domain viz cloud computing, mobile computing etc. These classes help us to assess vulnerabilities of any application belonging to one of these sub-domains. Theses classifiers can be modified when any software or hardware component is modified in a particular sub-domain . For example, we may classify Xen vulnerabilities in Cloud Computing $\rightarrow$ Amazon $\rightarrow$ IAAS $\rightarrow$ Hypervisor class as Amazon uses Xen as the hypervisor. If in later point of time, Amazon decides to use KVM as

hypervisor, we can update the classifiers accordingly. More details will be provided in the implementation section.

Once these classifiers are provided, the indexer creates an index with classes for the OKB. This index is referred by the SNLP module when a user performs a query. All the vulnerabilities matching user's application, technology and/or platform will be listed. User can then choose what vulnerabilities (s)he wants to test. Once the user selects the vulnerabilities he wants to test, necessary attacks are launched with the code from attack database. The wrapper scripts for the attack codes will provide necessary meta-data such as application path, necessary parameters. These attack scripts will launch attacks on the application and test it for the chosen vulnerabilities.

All the vulnerabilities tested positive will be reported to the user along with a security score based on the CVSS score. Necessary countermeasures will be provided if available. An illustration of our working framework is shown in Fig. 5.5. The implementation details for each component are detailed in the following section.

A typical use case scenario of using VULCAN components and modules to assess vulnerabilities for an android device using Mercury Framework [50] goes like this:

(1) A User provides both dynamic inputs for example "Android" (this data is provided to the System Classifiers module of our VULCAN framework), and a natural language query for example "Assess for weaknesses that can allow an unauthorized access to my device?" (this query is processed within our VULCAN Semantic Natural Language Processor - 'SNLP').

(2) The System Classifiers generates possible android based solutions and feeds them to the Indexer module. Then, the Indexer creates relevant vulnerabilities indexes which are used to produce vulnerabilities groups from the Vulnerability Class Index module. A sample created vulnerabilities group named "Root Access" contains indexed data of these CVE-IDs: CVE-2011-3874, CVE-2011-1823 and CVE-2009-2692.

(3) The SNLP component, will do reasoning tasks on the user query and using the

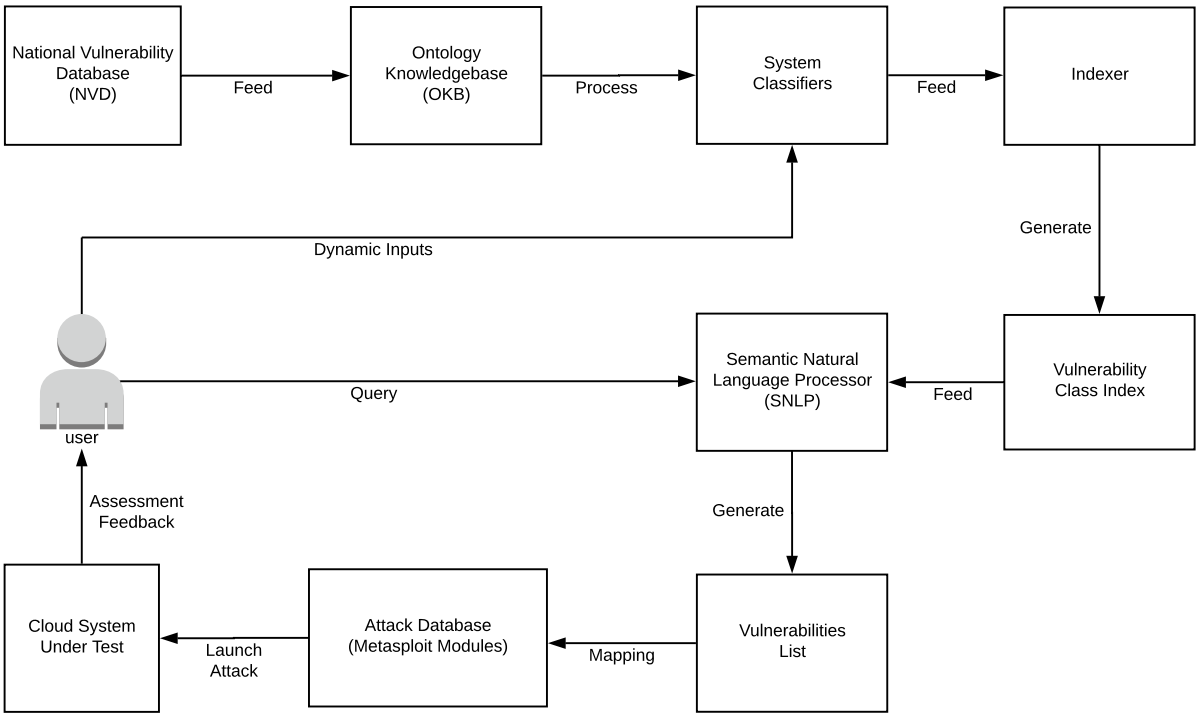

FIGURE 5.5. Vulcan Working

created vulnerabilities group data. It will return to the user via a dialogue agent interface relevant results such as the IT Products that have vulnerabilities and other necessary information that comply with the user query.

(4) Using our Middle-ware application, we map the IT Products to a Mercury framework [50] module called "Test for vulnerabilities that allow a malicious application to gain root access" to launch attacks on the products within our targeted android user device.

(5) Then, VULCAN traces the deployment of the module payloads and report whether the attacks were successful on the device or not and if the tested vulnerabilities are still present or fixed for those IT Products.

5.8. Vulcan Implementation

We have implemented our VULCAN via a set of interconnected components as described above in the Architecture section. The main source vulnerability information for our framework is provided by NVD. The NVD data is stored in a hierarchical database which lacks reasoning on its data. In our implementation of OKB we extract NVD data and store them in a graph database which is realized via Resource Description Framework (RDF) triples. With our graph database we generate an ontology that enable us to do some reasoning tasks which are useful for vulnerability assessment within our VULCAN.

To achieve a dynamic vulnerability assessment for Cloud Computing, we propose three modules such as: System Classifiers, Indexer, and Vulnerability Class Index. Each module depends on the other one as described in the Architecture section. In our SNLP implementation, we rely on our Ontology Knowledge Base for information and the capabilities of our modules to properly fetch the cloud computing relevant search results.

5.8.1. OKB

We defined a vulnerability ontology to model vulnerability information provided by NVD. In our approach, we extended the ontology proposed in our previous work on Vulnerability Assessment In Cloud Computing [79]. This new ontology in Fig. 5.6 is more expressive in terms of new entities and relationships, we added a class (CloudType) and sub-classes to help us model cloud environment and its types and also to model in the Software subclass of ITProduct class which vulnerable programs are privileged or unprivileged. We implemented this ontology in Protege [12] and the source code is available in our demonstration set samples.

Ontology Knowledge Base (OKB) Implementation is completed via these two steps:
(1) Extraction of vulnerability information from the National Vulnerability Database (NVD) - XML data feed.
(2) Population of our ontology knowledge base (OKB).

(1) Extraction of vulnerability information from NVD - XML data source

(a) Parsing the NVD - XML data source

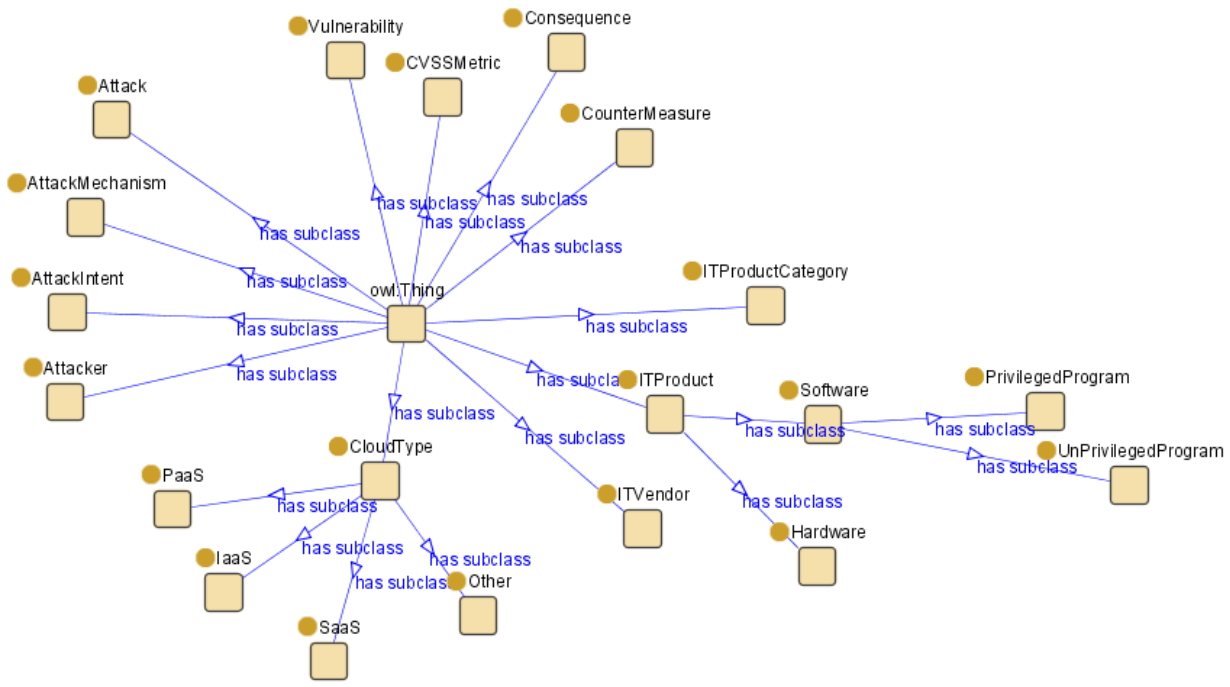

FIGURE 5.6. High Level View of our Vulnerability Ontology Definition

(b) Extracting from XML feed each entry relevant attributes for examples:

(i) CVE-ID

(ii) IT-Products

(iii) CVSS-Metrics

(iv) Summary

(c) From the extracted Summary text, another extraction take place to retrieve additional information (that

was not provided in any entry's attribute of NVD) about this vulnerability described in the Summary text.

such additional information are like:

(i) Who's the attacker

(ii) What's the attacker's intent

(iii) What's the attack mechanism

(d) Map the CVE-ID extracted in (b) to our web search agents to retrieve additional information about this

particular vulnerability. Such information are like:

(i) What is the attack (exploit)?

(ii) What is the consequence of the attack extracted in (c)?

(iii) What is the countermeasure of this attack (c)?

(2) Population of our OKB

(a) Using Protege-OWL editor [12], we first define our vulnerability ontology domain in terms of concepts (classes), roles(properties, relationships) and individuals [147].

(b) Then we populate our ontology to create a knowledge base of vulnerabilities. We use these two adopted approaches:

(i) Manually extract relevant vulnerability information from NVD - data source and use them to instantiate our ontology.

(ii) Using custom python script, we automatically extract relevant vulnerability information as described in Step-1.

(c) Then we store them into a triple store database. This database will be used to instantiate our ontology via Protege.

In the OKB process, we implemented our extractors using custom python scripts. These extractors, they iteratively retrieve relevant vulnerability information from each NVD entry. With the extracted data, we generate RDF triples using an RDFLIB [81] python library. With that we populated our defined ontology automatically. For a small set of NVD data entries, one can use Protege tool to achieve the same goal. By manually creating the ontology instances. In Protege, the ontology population can be achieved either by adding instances one at a time or by instantiating them using a backend database.

5.8.2. Modules

Our modules for the VULCAN implementation as illustrated in Fig. 5.7 are: System classifiers, Indexer and Vulnerability class index.

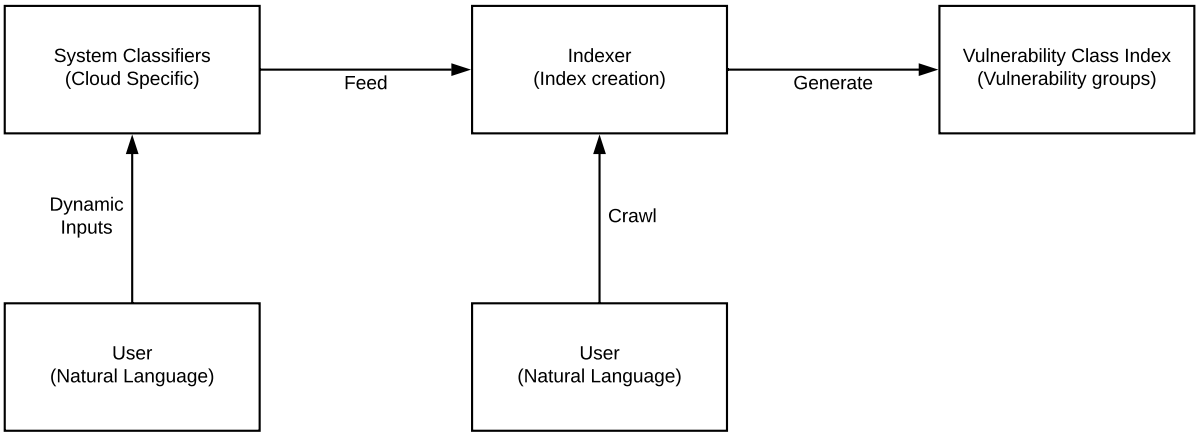

FIGURE 5.7. VULCAN Modules

5.8.2.1. System Classifiers

Our proposed approach for the system classifiers implementation is illustrated in our modules Fig. 5.7. We are customizing an application of genetic algorithms that are more adaptive to our dynamic inputs for cloud computing classification. Using the properties of genetic algorithms of working on a population of possible solutions and being stochastic, we rely on them on generating some classes that are then feed to our indexer module for further processing.

5.8.2.2. Indexer

The indexer module application will use the feeds received from system classifiers module to browse our vulnerability ontological knowledge base. As the module illustrates in Fig. 5.7, our indexer application should repeatedly check for any new change in the provided dynamic inputs and creates new indexes. Our goal for the implementation of this module is to optimize speed and performance in finding relevant information for the SNLP search queries.

We first collect the classes generated by the system classifiers, then use them to parse our OKB component into groups that are related to the provided feeds. Then we store the indexes as linked data. This approach will allow us to do inference on SNLP search results.

5.8.2.3. Vulnerability Class Index

The indexes created by our indexer module, are further processing and listed into vulnerability class groups as illustrated in Fig. 5.7. These groups reflect the cloud based dynamic inputs received. Then, within our VULCAN framework, the SNLP component uses these rich information about vulnerability for retrieving results for the relevant user given query. The implementation of this module is straight forward, all it needs to point an extractor to the indexed data and retrieve them as a list.

5.8.3. SNLP

Our Semantic Natural Language Processor engine enables users to search and reason about vulnerabilities via these interconnected modules:

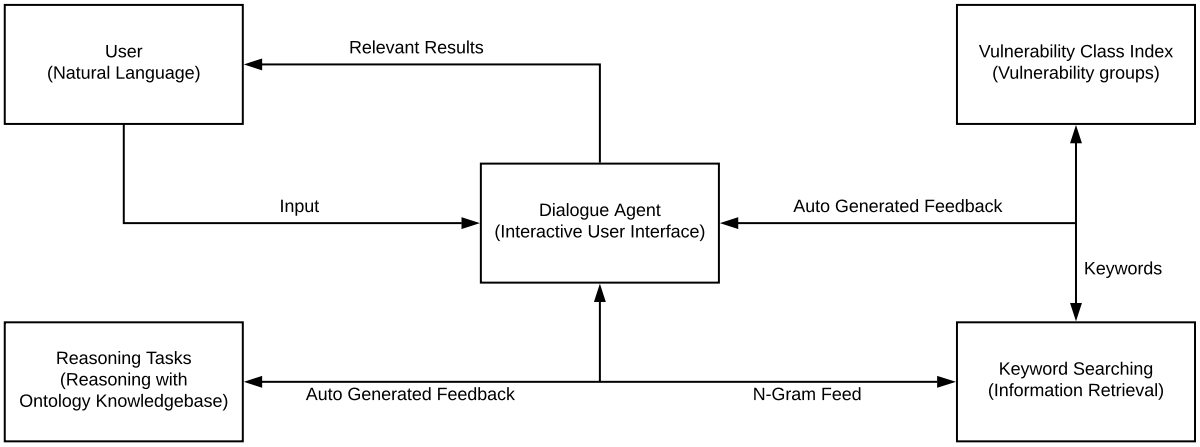

FIGURE 5.8. Vulcan Semantic Natural Language Processor

(1) pattern matching
   (a) This technique helps us to identify any kind of pattern from the user input text. We realize it using Regular Expression methods. Here we both do a keyword search and reasoning, then formalize a suitable result to respond the user query.
(2) keyword search
   (a) Queries protege plugin in [12] allows the user to query our vulnerability ontology using a plain keyword, or by selecting a class (concept) or relationships name

within our ontology. In addition to the search results, user to learn more about related information via generated inline links.

(3) reasoning

(a) To reason with our ontology knowledge base stored in an owl file for example, we use two methods. One method is by using the SWRL Protege plugin [107], here the user enter the SWRL rules via an editor to reason about owl individuals and to infer new knowledge about them. Another approach is to use Jess [6] as the rule engine to achieve the same goal as the SWRL plugin does.

(b) Pellet [39] is one of the reasoner tool we could use for OWL-DL [8] reasoning tasks.

(c) SPARQL query [11] allow us to query our RDF format ontology and perform some reasoning tasks.

Our SNLP component as illustrated in Fig. 5.8 is a self-contained application that allow the user to lively interact with a given system (in this case, our VULCAN) via a its dialogue agent interface. Here, the user input a query which can be a formal one (like a SPARQL query) or not. Then it is processed through our engine processor which run a pattern matching, keyword search and reasoning tasks while generating partial results. A formalized user query result is produced from one or a combination of the partial results. This application is implemented using a similar approach as the intelligent personal assistant and knowledge navigator system uses like in SIRI [10]. Here we interlock our OKB and modules (System Classifiers, Indexer, and Vulnerability Class Index) together to support our intelligent system. In order to be able to produce a reliable and relevant result of the user's query.

5.9. Future Work

In future we are planning to combine our ontology with the Ontology Of Cybersecurity Operational Information to provide a more robust and complete security in the cloud. The OVDB ontology primarily targets vulnerability of applications where as Ontology Of Cybersecurity Operational Information targets vulnerabilities of the cloud environment it-

self. Therefore, by combining these two ontologies we can achieve ontology for vulnerability assessment of the entire cloud infrastructure (application and environment). Since these two ontologies are cloud platform and application agnostic, we can perform vulnerability assessment for any application in any cloud.

5.10. Summary

In this chapter we have proposed and successfully implemented an Ontological Framework for Vulnerability assessment in cloud. The framework is capable of assessing the vulnerabilities in popular software as well as software created by users. The framework can be installed in any cloud platform and used for assessing any technology applications. The framework allows security professionals and as well normal users to search through the database and assess the software. The framework is equipped with a nice user interface which makes the searching of vulnerabilities very easy. The framework makes the tedious task of vulnerability management and assessment easy and effective. With vulnerabilities growing exponentially everyday, this framework will have a great use in present and future. As the framework is built with the state-of-art security automation protocols, it is both automotive and inter-operable with other applications.

CHAPTER 6

SECURE AND TRUSTED EXECUTION FRAMEWORK

In Chapters 3, 4, and 5 we have seen how to provide secure execution, provide on-demand integrity measurements, and efficiently manage vulnerabilities using ontologies in complex virtualized environments like Cloud Computing. In this chapter, we will see how we can combine these techniques to provide holistic trust and security to the entire lifecycle of workload execution.

6.1. Introduction

As modern day lives become more integrated with computing devices and services, we need secure and trusted computing systems. This necessity becomes even more critical due to the fact that most modern day information services involve remote data storage and remote computation. This trend of remote data storage and remote computation has accelerated due to:

(1) Advances in virtualization and related technologies.
(2) Widespread adoption of utility computing [34] delivered through the Cloud.
(3) Proliferation of mobile computing devices: smartphones, tablets, and laptops.
(4) Exponentially increasing IoT devices with limited resources.

These technologies and trends have improved the quality of life in general and made the benefits of utility computing easily accessible to the world at large. Along with these benefits come various challenges of trust and security. It is critical to ensure the security of this data and trust the programs accessing this data. By providing this guarantee, we can all take advantage of these technical solutions without the loss of data, services, and enable trustworthy business.

In this chapter, we will present a secure and trusted execution framework to achieve this goal. To provide holistic trust and security in a computing system, the framework offers principles that will provide trust and security guarantees throughout the execution life cycle

of the application. The framework is primarily designed and tested for virtualized environments with commodity PC hardware features. However, with the current proliferation of virtualization, we are confident that this framework can be deployed in any infrastructure and in any context.

The rest of this chapter is organized as follows: Section 6.2 will discuss the goals and design principles used for developing this framework. In Section 6.3, we will see the recommended architecture for this framework. To discuss the viability of the framework, Section 6.4 provides a detailed analysis of the framework and how it achieves the goals of this framework. Finally, Section 6.5 concludes the chapter with summary of contributions.

6.2. Overview

The primary objective of this framework is to ensure trust and security throughout the execution of an application. In other words, to provide a secure and trusted execution environment. We will be limiting our scope of security to confidentiality and integrity of the application and any security sensitive data of the application. The trust guarantees provided by the framework principles will apply to the infrastructure used to provide the execution environment along with the application. These trust guarantees are to ensure that only known, and by extension, trusted components are executing and providing the execution environment for the application.

6.2.1. Goals

As mentioned previously, the overall objective of this framework is to provide trust and security throughout the life-cycle of an application. An application's life cycle can be divided into three phases. If we can provide trust and security during these phases, we can ensure the trust and security guarantees are intact throughout the lifecycle of an applications. These phases and the goals are described below:

(1) At the time of launching the application:

We need to ensure that the application is launching in a trusted state. This would mean that the application itself is in a known state, free from any known vulner-

abilities, and is running in an environment that is in a known secure state. When the environment is in a known state, it will have all the components responsible for providing the execution in a trusted and verified state. To ensure that the environment is in a secure state, it should be free from any known vulnerabilities and also prevent any other internal/external component from interfering with the launch of the application.

(2) Any time during the execution of the application:

Once the application is launched in a secure and trusted state, it needs to continue executing in the same state. This would be more challenging than providing trust and security during the launch of an application because there will be more entities interacting with the application while it is executing compared to when it is launching. To provide these guarantees we need to ensure that:

(a) The application is continuing to be in a trusted state, especially before performing any security sensitive operations.

(b) All run-time interactions with the application are secure and the security controls are enforced by trusted components.

(c) The execution environment remains in a trusted state.

To provide the trust and security in an application any time during the execution of the application, we need to be able to fulfill these necessary conditions.

(3) At the time of updating the application:

Every application needs to be updated. It can be due to fixing the bugs, patching the vulnerabilities, or changing requirements. It is critical to ensure the trust and security guarantees for an application during this process of updating the application. It is important to note that when an application is being updated, the known or trusted state of the application changes. Because of this, there may also be changes in the application behavior and security requirements. Hence it is necessary to update the security controls and trust monitors of the application to reflect these changes. These controls and monitors need to be updated in a secure and

trustworthy manner.

6.2.2. Design Principles

To ensure holistic trust and security for an application during its entire execution lifecycle, the framework needs to be built over sound principles. These principles need to be derived from tested and validated practices. To ensure the necessary trust and security guarantees, our framework is designed on the basis of the principles learned from our previous works [80, 78, 105, 79, 72]. These principles are:

(1) **Only hardware can be implicitly trusted.**

Due to the dynamic and flexible nature of the software, it cannot be immutable and hence cannot be trusted *implicitly*. Hardware, when coupled with tamper-proof Read Only Memory (ROM) can be trusted implicitly. Only such a hardware component can be the initiator for the chain-of-trust. This property is the foundation for having trusted execution environment.

(2) **Trust can be extended from one entity to another entity.**

It has been shown in the past research [132, 36, 66] that trust in one entity can be extended to another entity. By extending the trust from one entity to another entity, we can build a chain-of-trust that can provide the necessary environment for executing an application in a trusted environment. Without this ability, we will need to compose a system with implicitly trusted components. While this may be feasible, such a system would be hard to build, test, and difficult to use.

(3) **Trust is extended to another component from a measuring entity if the measuring entity is trusted at the time of measurement.**

While it is possible to extend trust from one entity to another entity, it is important to follow the appropriate methodology. The trust from a measuring entity to the measured entity is extended only if the measuring entity is *trusted* at the time of measurement. Without this principle, the measuring entity itself will be untrustworthy and by extension, its measurement. This is why we need to start the chain-of-trust with an implicitly trusted entity.

(4) **Measurement process should be secure and tamper-proof.**

The foundation for trust is measurement. Measuring an entity's state and comparing it with a known trusted state will lead to trust in the application's state at the time of measurement. The process of measuring an entity and verifying the trustworthiness of its state should be secure and tamper proof. This will prevent malicious programs and actors spoofing incorrect measurements to wrongly trust the application's state.

(5) **Measurements need to be stored in and reported by trusted hardware.**

After securely measuring an entity, the measurement need to be stored in trusted hardware. This will guarantee that the measurements will not be tampered with for a later usage. A similarly trusted hardware should be used to report the measurement to a remote or local verifier. After storing the measurement in the trusted hardware, it needs to be ensured that the measurement is reported to the verifier in a secure manner. This would include secure communication channel and protocols for communicating with the verifier.

(6) **All interactions between the trusted components need to be secure.**

Once trust is established in various components, it is important to maintain the trust. If these components don't interact with any other entities and no other entities interact with these components, the trust remains same. However, such a system will not be practical to build and use. Any reasonably functional system's components will be interacting with each other. All such interactions need to be secure to maintain the trustworthiness of the verified components.

(7) **All components of the system need to be free from known vulnerabilities.**

Even when a system is composed of entities in known trusted state and it is ensured that all interactions with the components are secure, it would not guarantee that the known states are secure by themselves. Also, it is not practical to know or identify all weaknesses in the application at any given time, the best we can expect

to do is to ensure that all components are free from known vulnerabilities, and in doing so we can ensure the system is in the best possible secure and trusted state.

6.3. Architecture

In this section, we will present the proposed architecture of our framework and discuss each component in detail. The framework is separated into layers to provide better visualization and clarity on the functionality in each layer. Figure 6.1 shows the diagrammatic representation of the architecture.

All trusted components are colored in Green and untrusted components are colored in Red. All the components that are not directly part of the execution environment are colored in Orange. Whenever there is a measurement by a trusted entity, such a measurement is represented with Blue arrows. All other interactions and components are depicted with black boxes or black lines.

6.3.1. Hardware Layer

The most important layer in the framework is the hardware layer. It provides the foundation for trust and by extension, provides security guarantees for the rest of the framework. It consists of hardware components necessary to provide the trusted and secure execution environment.

6.3.1.1. Immutable Root of Trust for Measurement (IRTM)

IRTM is the key hardware component that provides the implicit trust to build the chain-of-trust. As the name implies, it is immutable and consists of Read Only Memory (ROM) and can be implicitly trusted. The ROM has code necessary for measuring the next component in the chain-of-trust to be executed. The IRTM executes very early in the system startup and validates all the firmware and system software to be executed. The IRTM has direct access to the next components in the system's memory or in another hardware component and is secured by the platform. The platform ensures that no other software or firmware is executing when IRTM is measuring. Due to this feature, IRTM should usually be

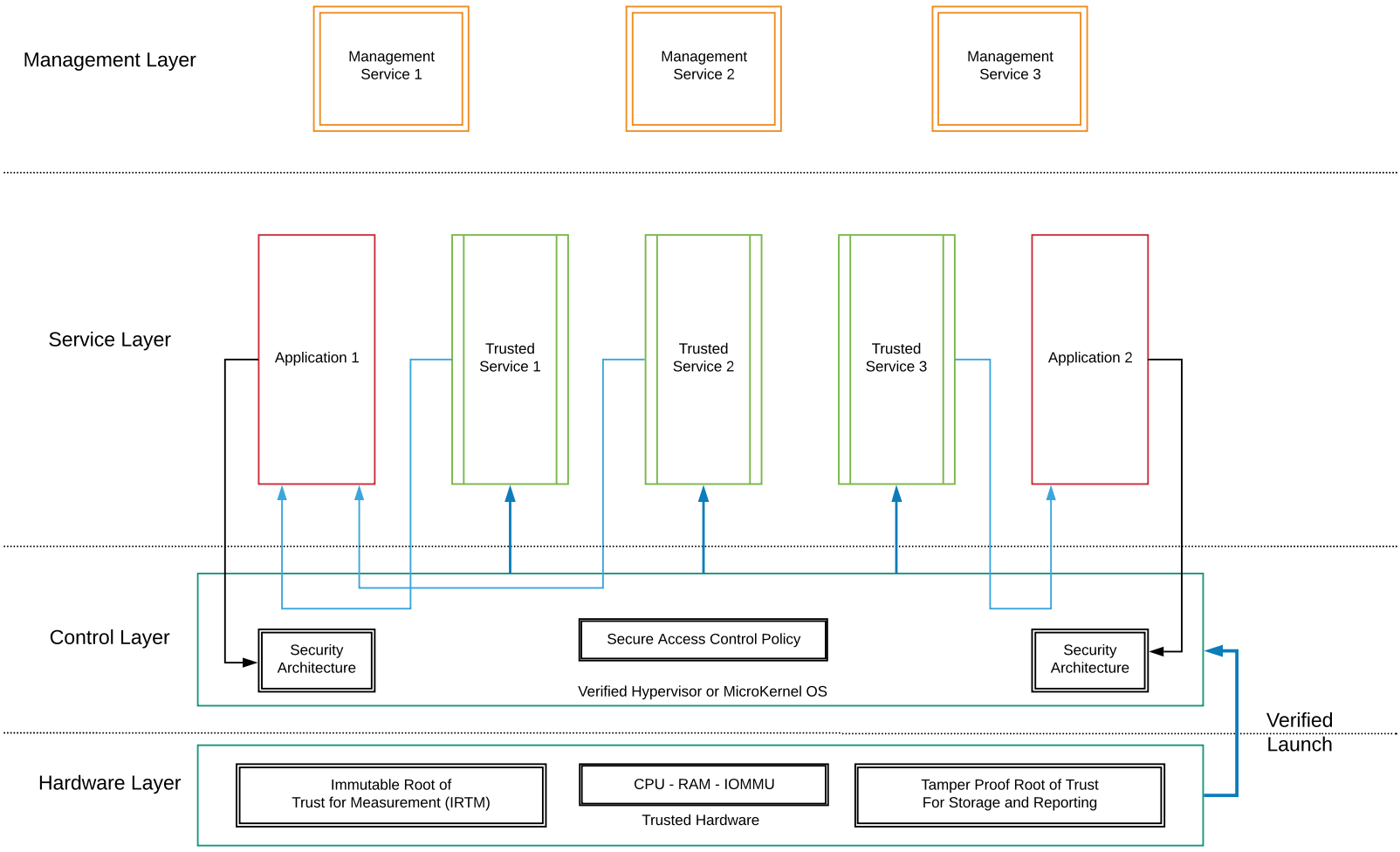

FIGURE 6.1. Secure and Trusted Framework - Architecture

executing during the the system startup or rest. Also, such an isolated execution guarantees the security of the measurement being performed.

6.3.1.2. Processing Subsystem

The Processing Subsystem consists of typical processing components: CPU, RAM, and I/O Memory Unit.This subsystem is responsible for executing all the software components, such as, hypervisor, operating system, and applications. There are no special requirements for the processing subsystem. Any off the shelf commodity components would suffice. However, it is necessary for them to provide modern virtualization features. CPU,

Memory, and I/O virtualization.

6.3.1.3. Root of Trust for Storage and Reporting

With so many measurements being performed, we would need a trusted storage mechanism along with trusted reporting capabilities. It is ideal to implement these functions in hardware rather than the software layer. This root-of-trust would allow any component in the architecture to store measurements. While this flexibility may seem like a weakness, it should not. Because all the trusted components only verify measurements performed by them. Additionally, when the measurement is reported to the verifier, entire platform state gets reported, including who performed the measurements. This will alleviate any concerns of spoofing.

6.3.2. Control Layer

The next layer in the architecture is the control layer. This layer consists of a system software, typically hypervisor or a microkernel [156, 62], capable of virtualizing operating systems and applications. This layer has complete control over the underlying hardware layer and can delegate access to the hardware resources to any component necessary. This layer should ideally have a small codebase and is preferably formally verified. These constraints will greatly reduce the security concerns and by extension, the need to verify the state of the control layer software for every interaction.

Furthermore, the control layer will delegate most runtime operations to other components and is only concerned about allocating resources and monitoring secure access control. It uses the modern virtualization techniques for memory, CPU, and I/O virtualization to achieve this. This greatly shrinks the attack surface during the runtime and the potential for attacks. With this mechanism, we can continue to trust the control layer even for extended times.

To maintain the security of the system, it is necessary to have an access control mechanism. An access control mechanism should be present in the control layer to monitor the interactions between the workloads. Attribute Based Access Control (ABAC) [64, 63] is

best suited for providing access control between the various components of this architecture. ABAC would be able to take the current state of a component (as an attribute) to validate whether the requested access can be granted or not.

Finally, the control layer provides security architectures for secure applications. These security architectures are implemented by using modified virtual hardware [78, 105] to provide security features to the applications. These security features range from protecting secure keys to integrity verification of the entire memory. At the same time, these architectures are capable of providing varying degrees of security at various granularities. The smallest granularity that can be provided is one address location in the memory.

6.3.3. Service Layer

Next up in the stack is the Service Layer. Service layer consists of measuring services used by control layer and the workloads. The service layer also consists of the workloads. Workloads can be an operating system, or minimal kernel based applications like unikernel [86], or simply applications [92] running in an isolated environment. All service layer components are isolated by a trusted control layer.

Measuring services themselves are technically workloads executing on top of the control layer. Their functionality can be as basic as measuring the state of other workloads or external entities like management services. Measuring services are basically agents of the control layer and can be executed on demand to perform various measurements at any time.

All measuring services are brought up, managed, and brought down by the control layer. These services are launched only after being verified by the control layer that they are in a trusted state. These are usually ephemeral in nature and are brought up as and when necessary. After they are launched, they only exist for providing a specific service and interact with limited entities with limited functionality. All the additional capabilities and control of accesses to and from the services are monitored by the access control mechanism in the control layer.

6.3.4. Management Layer

All applications and workloads will change through time. It can be due to changes in the requirements or it can be due to newly discovered bugs and vulnerabilities. To manage these changes effectively, the management layer provides necessary services. Management layer is the most external layer in the architecture. It is not directly part of the execution environment, but is necessary for maintaining the security of the execution of the workload.

Management layer will provide services external to the workload execution that can detect, protect, and modify the workload. For performing any changes to the state of the workload, a management layer service has to work with the measuring service. The measuring service will ensure that the management service is trustworthy and the process of the changing or updating a workload's state is secure. The measuring service will also be responsible for updating the access control rules to reflect these changes.

6.4. Analysis

In this section, we will present a detailed analysis of our framework. We are primarily focused on the trust and security of the execution and hence will not discuss the performance in terms of resource consumption. This section is discussed in 3 subsections. We will first provide an analysis of the possible implementation of our architecture. Then we will deep dive into the trust and security of the architecture. We will present how the trust and security is provided in the 3 crucial stages of a workload's execution. Lastly, we will discuss the limitations and potential challenges to our proposed framework.

6.4.1. Implementation

A reference implementation is presented in Figure 6.2 which shows what technologies and services are possible in our architecture. While existing technologies don't exactly map to the recommended specifications, we believe they can match the architecture in principle. We will discuss how various components can be implemented with existing technologies and what improvements are necessary to achieve the ideal implementation.

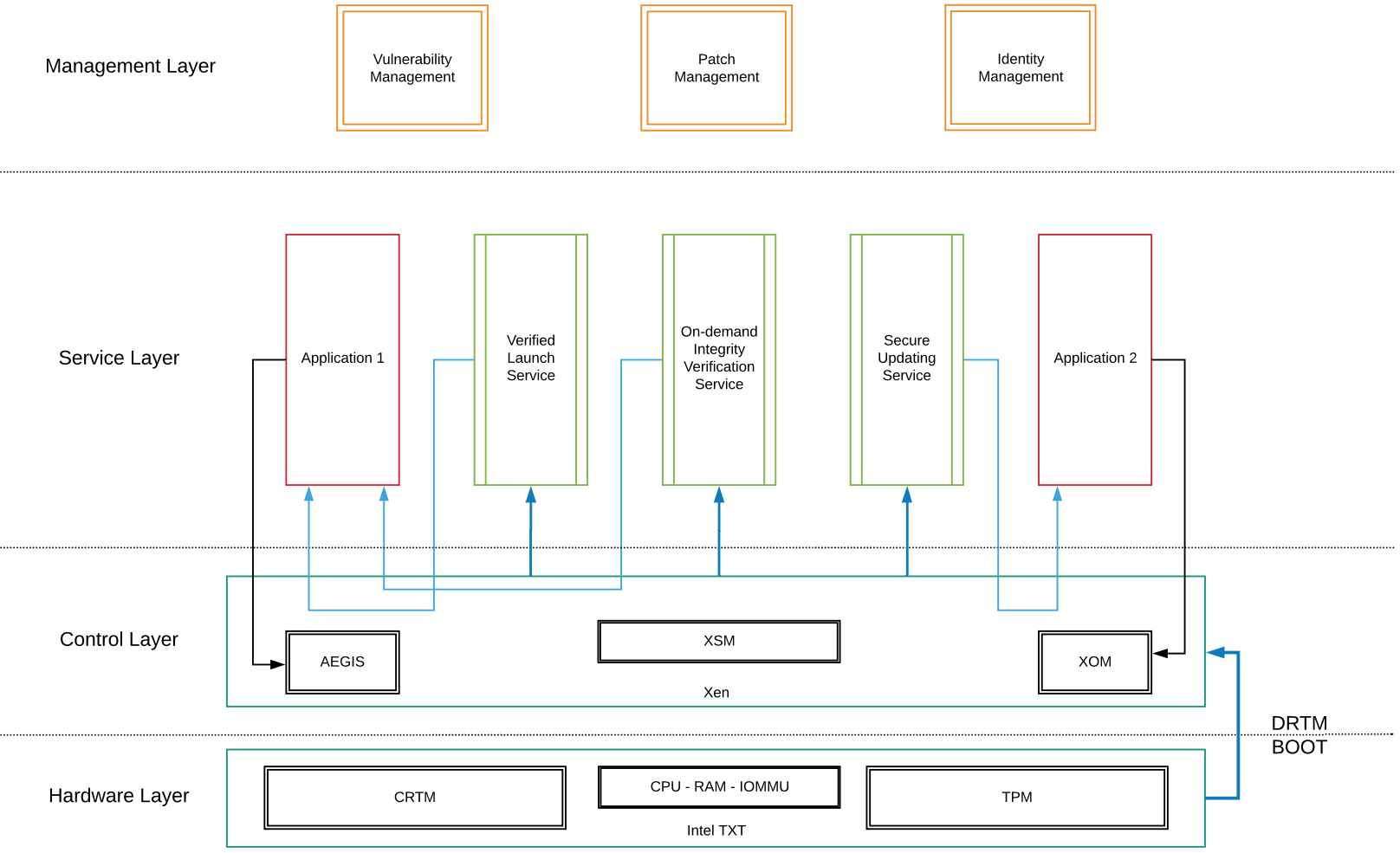

FIGURE 6.2. Secure and Trusted Framework - Implementation

6.4.1.1. Hardware Layer

The hardware layer has 3 main components. We believe all these 3 components can be implemented using currently available commodity hardware. To provide IRTM, we can reuse the existing CRTM in Intel [67] and AMD [127] platforms. These platforms also provide instructions `GETSEC[SENTER]` and `SKINIT` respectively to provide an isolated execution environment and launch the control layer. This isolation is achieved by pausing all other threads of execution and only executing the control layer. The control layer can be launched into a trusted state either at the time of boot or at any other time dynamically.

With this ability to launch/reset the control layer into a trusted state at any time is extremely necessary as the trust in the underlying trusted hardware will now be extended to the control layer. And the control layer will proxy as the root-of-trust for performing dynamic on-demand measurements for the remaining components in the system.

6.4.1.2. Control Layer

The control layer needs a system software capable of virtualizing the workloads. There are few virtual machine monitors and microkernels that are capable of doing this. Some of the microkernels are formally verified [76, 28] and are capable of running full operating systems, driver domains, or applications. One research hypervisor is also formally verified [20] and it has been proven that it is also possible to formally verify a regular hypervisor [82].

Due to other functionalities required to be performed by the control layer, we believe a small, modular mainstream hypervisor like Xen [26] would be more ideal for current implementation of our framework. Xen's modular architecture allows us to run most of the components in isolated address spaces [52] like driver domains. With this architecture, Xen behaves more like a microkernel [62].

Along with modular architecture and strong isolation, xen also implements access control mechanism modeled after widely used SeLinux [90]. With the presence of this access control mechanism, Xen can ensure security during any interactions between the workloads. This also allows xen to provide measuring services which are capable of performing measurements with fine granularity.

6.4.1.3. Service Layer

As the measuring services are specialized workloads, designed for providing measurements for the control layer and other workloads as necessary, they can be implemented by customizing the workloads. For example, virtualization specific minimal os [101], unikernels [87] or even applications with full blown operating system can be used to provide a measuring service. If the measuring service has minimal TCB, it will have better performance and security guarantees. Solutions like Qemu Lite PC [89], a virtual machine type

optimized to run minimal operating systems in virtualized environments, are much better suited for running the measuring services. The measuring services are themselves launched after being verified by the control layer.

A wide variety of services can be implemented in the service layer. We will present few key services that we have implemented in the Radium [80] architecture:

(1) *Trusted Launch:*

Trusted Launch measuring service is responsible for measuring the state of a workload image and launch only if it is in a trusted state. To measure the state of a workload, we used the SHA256 hash of the workload (virtual machine) image and verified it against a known good value of the workload.

(2) *On-demand measurement:*

After launching the workload, the on-demand measurement service is responsible for performing anytime on-demand measurements to verify the state of the workload dynamically. This would allow users to verify the state of the workload before performing any security sensitive transaction.

(3) *Malware detection:*

We have also implemented a malware detection service. This service can detect if the workload has been infected by a specific malware. Since the measuring services can have access to the workload's (guest) physical memory, we can perform the measurement without depending on the workload's operating system, if it has one. We were successfully able to detect the kbeast [35]. This detection would have been extremely difficult if performed from within the workload itself because kbeast is a stealthy rootkit.

6.4.1.4. Management Layer

As previously discussed, the management layer consists of services that are necessary to maintain the security of the workload's execution. Vulnerability management is one such service that is extremely important to ensure that the workload is secure. Robust vulnerability management practice helps us to test the entire stack for any potential vulnerabilities

that can threaten the security of the workload. We have implemented such a service in Vulcan [72] extending our prior work OVDB [79]. Other potential services that can potentially be implemented in the management layer are Patch Management and Identity & Access Management (IAM). Patch management would work in tandem with the vulnerability management and will be responsible for updating the workload to a more secure (less vulnerable) state. IAM would be responsible for ensuring only authenticated users can only access system components that they have authorization for.

6.4.2. Security and Trust

To analyze the security and trust provided by the proposed framework, we will discuss security and trust in 3 phases of the workload's execution. Before analyzing these three phases, we need to be aware of what components are available and which of these components are trustworthy. The hardware and control layers are present before any stages of the workload execution begin.

The hardware layer is implicitly trusted.The hardware is responsible for launching the trusted control layer. The original trust in the control layer is derived from the verified launch by the trusted hardware (IRTM). After the IRTM measures and verifies that the control layer software is in a known state, the control layer starts executing and assumes control over the entire hardware. The trust in the security architectures is derived from the trust in the control layer. While they logically appear in the control layer, they are to be implemented as plugins or extensions to the control layer software. Through this, an additional layer of isolation is achieved and core control layer software would be protected from any security challenges against the security architectures themselves.

6.4.2.1. At the Time of Launch

Trust: A workload is always launched by the trusted control layer. The control layer can use the verified launch measuring service to ensure that the workload is in a trusted state as necessary. This ensures that only a trusted workload is launched. At the time of launch, the hardware and control layers are completely trusted. An additional integrity verification

of the control layer can be requested if necessary for the workload. This would guarantee that the workload is launched into a verified trusted environment.

Security: The control layer will use advanced hardware virtualization features to provide a securely isolated execution environment. Since this isolation is provided by a small TCB code, it has The control layer's access control policy prevents any other component to interfere with the launch of the workload. Thus, the workload will be securely launched into an isolated environment.

6.4.2.2. Any Time During the Execution

Trust: The control layer is primarily responsible for ensuring all the components that are performing measurements or security sensitive operations are in a trusted state. While the workload is executing, we may need to verify the trustworthiness of the workload before performing any security sensitive transaction. For example, the workload need to be in a trusted state before accessing DRM content. In such scenario, the state of the workload can be reliably determined by the associated measuring service and access to the DRM content will be provided only if the workload is in a trusted state.

Similarly, if the workload needs to send information only to a trusted component, the component's state will be verified before passing such information. For example, the workload needs to send a sensitive transaction over the network interface only if the network driver is trustworthy. It is important to note that the trust and security guarantees of the framework are available only within the execution environment. Once the data passes the execution environment (like a network packet in the above example) the trust and security is dependent on the medium of transmission and the receiving endpoint.

Security: The workload's runtime security is primarily provided by the access control mechanism in the control layer and security architecture (if any is used by the workload). The access control mechanism prevents any and all interactions with the workload from other components. Especially the measuring services, as they can potentially have ability to access the (guest) physical memory of the workload through the control layer.

The execution environment is secured as a whole unit by the access control mech-

anism not the individual threads of execution within the workload. To have more finer security features, the workload needs to rely on the security architecture. A workload can have a custom security architecture built for itself or can use one of the existing security architectures. By using a security architectures, the workload can request protection for the security sensitive components at a page granularity. This facility is available even for an application within the Operating System workloads. The security architecture can prevent the unauthorized accesses from even the operating system or rootkits within the workload.

6.4.2.3. At the Time of Change/Update

Trust: Managing changes or updating the workload is another crucial operation facilitated by the framework. A measuring service will mediate with the management services to ensure that the workload is only interacting with trusted management services. This can be performed by validating the digital certificate of the management service and validating the state of the platform the management service is executing upon. Once the trust in the management service is established, the workload will be permitted to initiate the update process.

Security: Security of the updating process is as critical as the trust in the components performing the update. The access control policy will be evaluating if the management services have the necessary access to perform scans to ascertain if the workload needs an update. Certain types of scans can be intrusive than the others. Relevant access control policies need to be added to allow management services to perform necessary scans. Once the management service ascertains the need the for an update, the access control policy will only allow the appropriate service to perform the update. For example, only the vulnerability management service will be allowed to perform approved scans and the patch management service will be allowed to perform the patching.

6.4.3. Challenges

In this subsection, we will highlight potential challenges and limitations of our proposed framework.

(1) Physical attacks: If the trusted hardware is replaced with a malicious hardware when the system is offline, any new workloads installed can be compromised. This would require physical access to the hardware and can be detected or prevented using physical security measures.

(2) Performance issues: All the new operations added for ensuring trust and security, will result in non-negligible performance overhead. The actual overhead varies from workload to workload and the type of measuring services or security architectures it requires. There is also potential lag created when the system waits for the verification to be performed using hardware reporting. This can affect the usability of the system.

(3) Designing applications for security architectures: Designing or modifying applications to work with security architectures provided by the framework may not be straight forward. This may complicate the development, deployment, and management of the regular operation of the application.

(4) Reliability of the security architectures: When the security architectures are implemented, some of the security benefits are easily discernible. It is important to understand the security guarantees provided by the security architecture and proper attention needs to be paid in selecting the appropriate architecture for a specific workload.

(5) Unknown vulnerabilities: The vulnerability management service can only protect the workload against known vulnerabilities. A zero day vulnerability in any component of the stack, can compromise parts of the trust and security guarantees provided by the framework. This challenge is common for all security solutions.

(6) Unknown backdoors (Intel ME): Not all the features of the hardware or firmware are always open and well known. Some of these features can be surprisingly vulnerable to a determined attacker. It is quite impossible to defend against a hidden feature that can be leveraged as an undetected backdoor into the system. This problem affects any solution working on a hardware with hidden backdoors and is not specific

to our framework.

6.5. Summary

In this chapter, we have presented the necessary principles required to build a secure and trusted execution environment for virtualized workloads. We also provided a reference architecture detailing all the necessary components and their operations. A thorough analysis of our framework is provided to understand how to implement the architecture, trust and security provided by the architecture, and various challenges to consider when adopting this framework.

CHAPTER 7

CONCLUSION

7.1. Contributions

The first contribution of our work is Virtualization Based Secure Execution (vBASE) framework. This framework enables implementation, testing, and executing security architectures in the virtualization layer. By implementing security architectures in the virtualization layer, vbase can provide confidentiality and integrity for the virtualized workloads. We have implemented a simple prototype to demonstrate the framework's abilities. In this example we have shown that a secure application can directly communicate with the hypervisor bypassing the operating system and protect security sensitive information from powerful attackers. The prototype example was successful in preventing the overwriting of secure keys by the Operating System.

The second contribution of our work is Race-free On-demand Integrity Measurement (Radium) Architecture. This architecture eliminates the TOCTTOU attacks by providing on-demand measurements. To provide trustworthy integrity measurements on demand, we have implemented Asynchronous Root of Trust for Measurement (ARTM) by extending DRTM. ARTM extends the trust in the platform to the hypervisor using DRTM's verified launch. By keeping the hypervisor TCB and runtime interactions minimal, we have extended the trust in the hypervisor beyond the verified launch. By taking advantage of the ARTM, we designed measuring services that can perform trustworthy on-demand measurements and advanced security solutions such as a rootkit detector. In our prototype implementation, we have demonstrated that our platform can perform trustworthy dynamic measurements and detect stealthy rootkits.

The third contribution of our work is Ontology based vulnerability management which is powered by an ontology based vulnerability database (OVDB). This database includes vulnerabilities, exploits, and mitigations. By modeling all these databases using ontologies, we can connect all related concepts logically without using queriesand demonstrated that

it is more efficient and intuitive. It also allows use to use logical reasoning and semantic natural language processing in working with vulnerabilities.

We have further extended our OVDB by implementing an ontology based vulnerability assessment for cloud computing (Vulcan) and proven that ontology based vulnerability management can perform well with complex software stacks such as cloud computing.

Our final contribution is the Secure and Trusted Execution Framework for Virtualized Workloads. This framework provides trust and security guarantees to virtualized workloads throughout their execution lifecycle. We have provided design principles necessary for ensuring trust and security execution based on our previous works. This framework can be implemented using modern virtualization hardware and the Trusted Platform Module (TPM), without the need of expensive custom hardware. Along with the design principles we have provided detailed analysis on implementation strategies and proven that our framework actually provides trust and security guarantees throughout the execution lifecycle of virtualized workloads.

7.2. Future Directions

With continued adoption of cloud computing, information services are becoming increasingly distributed in nature. Most of the data and computation is hosted on remote computers. In this light, it is important to have secure and trustworthy systems. While large scale formal verification of systems may be difficult in the foreseeable future, it is definitely possible to build trusted and secure computing systems with a small verified systems-software. Research in this direction will help us realizing the vision of trusted and secure computing.

APPENDIX

AUTHOR BIOGRAPHY

Srujan Kotikela received his B.Tech degree in Computer Science and Engineering from Jawaharlal Nehru Technological University, Hyderabad, India, in 2008. He received his M.S. degree in 2014 and is currently a Ph.D. Candidate in the Department of Computer Science and Engineering at the University of North Texas. His research interests are in the areas of computer security, trusted computing, and cloud computing with emphasis on virtualization based security solutions.

Publications

(1) **Srujan Kotikela**, Tawfiq Shah, Mahadevan Gomathisankaran, Gelareh Taban (2015) Radium: Race-free On-demand Integrity Measurement Architecture In: International Conference on Privacy, Security, Risk and Trust (PASSAT) ASE.

(2) **Srujan Kotikela**, Mahadevan Gomathisankaran (2013) Work In Progress : Privacy Against Unlawful Surveillance (PRIUS) In: Annual Computer Security Applications Conference (ACSAC), New Orleans, Louisiana.

(3) Patrick Kamongi, **Srujan Kotikela**, Mahadevan Gomathisankaran, Krishna Kavi (2013) A methodology for ranking cloud system vulnerabilities In: 2013 Fourth International Conference on Computing, Communications and Networking Technologies (ICCCNT),1-6.

(4) Patrick Kamongi, **Srujan Kotikela**, Krishna Kavi, Mahadevan Gomathisankaran, Anoop Singhal (2013) VULCAN : Vulnerability Assessment Framework for Cloud Computing In: 2013 IEEE 7th International Conference on Software Security and Reliability (SERE), 218-226.

(5) Satyajeet Nimgaonkar, **Srujan Kotikela**, Mahadevan Gomathisankaran (2012) CTrust : A framework for Secure and Trustworthy application execution in Cloud computing Academy of Science and Engineering (ASE) Science Journal 1: 4. 152-165.

(6) **Srujan Kotikela**, Krishna Kavi, Mahadevan Gomathisankaran (2012) Vulnerability Assessment in Cloud Computing In: The 2012 International Conference on Security & Management (SAM 2012) 67-73 CSREA Press.

(7) Satyajeet Nimgaonkar, **Srujan Kotikela**, Mahadevan Gomathisankaran (2012) CTrust : A Framework for Secure and Trustworthy Application Execution in Cloud Computing In: 2012 International Conference on Cyber Security (CyberSecurity), 24-31.

(8) **Srujan Kotikela**, Satyajeet Nimgaonkar, Mahadevan Gomathisankaran (2011) POSTER : Virtualization Based Security Framework (vBASE) In: Annual Computer Security Applications Conference (ACSAC), Orlando, Florida.

(9) **Srujan Kotikela**, Satyajeet Nimgaonkar, Mahadevan Gomathisankaran (2011) Virtualization Based Secure Execution and Testing Framework In: 7th International Association of Science and Technology for Development (IASTED) Parallel and Distributed Computing Systems, Secretariat, B6, Suite 101, 2509 Dieppe Ave. SW, Calgary, AB, Canada T3E 7J9: ACTA PRESS.

REFERENCES

[1] *Xen Cloud Platform*, `http://www.xen.org/download/xcp/index_1.1.0.html`.

[2] *Common platform enumeration*, 2012.

[3] *Common vulnerabilities and exposures*, 2012.

[4] *Common vulnerability scoring system*, 2012.

[5] *Common weakness enumeration*, 2012.

[6] *Jess,the rule engine for the java platform*, 2012.

[7] *National vulnerability database*, 2012.

[8] *Owl web ontology language guide*, 2012.

[9] *Security content automation protocol*, 2012.

[10] *Siri*, 2012.

[11] *Sparql query language for rdf*, 2012.

[12] *welcome to protege*, 2012.

[13] *Cvss severity over time*, 2017.

[14] *Notpetya ransomware*, 2017.

[15] *Petya ransomware*, 2017.

[16] *Wannacry ransomware*, 2017.

[17] *Yahoo data breach*, 2017.

[18] F. Abdoli and M. Kahani, *Ontology-based distributed intrusion detection system*, Computer Conference, 2009. CSICC 2009. 14th International CSI, oct. 2009, pp. 65 –70.

[19] Keith Adams and Ole Agesen, *A comparison of software and hardware techniques for x86 virtualization*, ACM SIGARCH Computer Architecture News 34 (2006), no. 5, 2–13.

[20] Eyad Alkassar, Mark A Hillebrand, Wolfgang Paul, and Elena Petrova, *Automated verification of a small hypervisor*, International Conference on Verified Software: Theories, Tools, and Experiments, Springer, 2010, pp. 40–54.

[21] Eyad Alkassar, Mark A. Hillebrand, Wolfgang J. Paul, and Elena Petrova, *Automated*

verification of a small hypervisor, Proceedings of the Third International Conference on Verified Software: Theories, Tools, Experiments (Berlin, Heidelberg), VSTTE'10, Springer-Verlag, 2010, pp. 40–54.

[22] AMD, *AMD64 virtualization codenamed "Pacifica" technology — secure virtual machine architecture reference manual*, Tech. Report Publication Number 33047, Revision 3.01, AMD, May 2005.

[23] AMD, *Technology (iommu) specification*, (2007).

[24] Ashish Thakwani, *Process-level Isolation using Virtualization*, `http://repository.lib.ncsu.edu/ir/handle/1840.16/2031`, 2010.

[25] John Banghart, Stephen Quinn, and David Waltermire, *Open vulnerability and assessment language (oval) validation program test requirements (draft)*, US Department of Commerce, National Institute of Standards and Technology, 2010.

[26] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, and Rolf Neugebauer, *Xen and the art of virtualization*, Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03) (Bolton Landing, NY, USA), ACM, October 2003, pp. 164–177.

[27] Erick Bauman, Gbadebo Ayoade, and Zhiqiang Lin, *A survey on hypervisor-based monitoring: approaches, applications, and evolutions*, ACM Computing Surveys (CSUR) 48 (2015), no. 1, 10.

[28] Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer, *Formal verification of a microkernel used in dependable software systems*, International Conference on Computer Safety, Reliability, and Security, Springer, 2009, pp. 187–200.

[29] Sushil Bhardwaj, Leena Jain, and Sandeep Jain, *Cloud computing: A study of infrastructure as a service (iaas)*, International Journal of engineering and information Technology 2 (2010), no. 1, 60–63.

[30] M. Bishop and D. Bailey, *A critical analysis of vulnerability taxonomies*, 1996.

[31] Jeramiah Bowling, *Virtual security: Combating actual threats*, Linux J. 2011 (2011), no. 205.

[32] Micah Bushouse and Douglas Reeves, *Hyperagents: Migrating host agents to the hypervisor*, Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy, ACM, 2018, pp. 212–223.

[33] John Butterworth, Corey Kallenberg, Xeno Kovah, and Amy Herzog, *Bios chronomancy: Fixing the core root of trust for measurement*, Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (New York, NY, USA), CCS '13, ACM, 2013, pp. 25–36.

[34] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic, *Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility*, Future Generation computer systems 25 (2009), no. 6, 599–616.

[35] Richard Carbone, *Malware memory analysis of the kbeast rootkit: Investigating publicly available linux rootkits using the volatility memory analysis framework. scientific report*, Scientific Report (in preparation for publishing). Defence R&D Canada–Valcartier (2014).

[36] David Challener, Kent Yoder, Ryan Catherman, David Safford, and Leendert Van Doorn, *A practical guide to trusted computing*, first ed., IBM Press, 2007.

[37] B. Chandrasekaran, John R. Josephson, and V. Richard Benjamins, *What are ontologies, and why do we need them?*, IEEE Intelligent Systems 14 (1999), no. 1, 20–26.

[38] Joseph Cihula, *Trusted boot: Verifying the xen launch*, Fall 2007.

[39] Clark and Parsia, *Pellet reasoner plug-in for protege 4*, 2012.

[40] L Corrons, *Pandalabs annual report*, Tech. report, Technical report.–PandaLabs, 2017.

[41] Victor Costan, Ilia Lebedev, Srinivas Devadas, et al., *Secure processors part i: Background, taxonomy for secure enclaves and intel sgx architecture*, Foundations and Trends® in Electronic Design Automation 11 (2017), no. 1-2, 1–248.

[42] ———, *Secure processors part ii: Intel sgx security analysis and mit sanctum architecture*, Foundations and Trends® in Electronic Design Automation 11 (2017), no. 3, 249–361.

[43] Mache Creeger, *Cloud computing: An overview.*, ACM Queue 7 (2009), no. 5, 2.

[44] A Crespo, I Ripoll, M Masmano, P Arberet, and JJ Metge, *Xtratum an open source hypervisor for tsp embedded systems in aerospace*, Data Systems In Aerospace DASIA, Istanbul, Turkey (2009).

[45] Michael Cusumano, *Cloud computing and saas as new computing platforms*, Communications of the ACM 53 (2010), no. 4, 27–29.

[46] Johan De Gelas and Intel ESX, *Hardware virtualization: the nuts and bolts*, AnandTech. Retrieved March 17 (2008), 2008.

[47] Dick OBrien, *Ransomware 2017*, `https://www.symantec.com/content/dam/symantec/docs/security-center/white-papers/istr-ransomware-2017-en.pdf`, 2017.

[48] Jeffrey Dwoskin, Mahadevan Gomathisankaran, and Ruby Lee, *Framework for design validation of security architectures.*

[49] Thomas D East, CJ Wallace, AH Morris, RM Gardner, and Dwayne R Westenskow, *Computers in critical care.*, Critical care nursing clinics of north america 7 (1995), no. 2, 203–217.

[50] T. Erasmus, *The heavy metal that poisoned the droid*, (2012).

[51] Dieter Fensel, *Ontologies*, Ontologies, Springer, 2001, pp. 11–18.

[52] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, Mark Williamson, et al., *Safe hardware access with the xen virtual machine monitor*, 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS), Boston, USA;, 2004, pp. 1–1.

[53] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh, *Terra: A virtual machine-based platform for trusted computing*, Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (New York, NY, USA), SOSP '03, ACM, 2003, pp. 193–206.

[54] Matt Gillespie, *Best practices for paravirtualization enhancements from intel virtualization technology: Ept and vt-d*, Retrieved November 26 (2009), 2013.

[55] Robert P Goldberg, *Survey of virtual machine research*, Computer 7 (1974), no. 6, 34–45.

[56] M. Gomathisankaran and A. Tyagi, *Arc3d : A 3d obfuscation architecture*, High Performance Embedded Architectures and Compilers (HiPEAC) (2005), 184–199.

[57] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir, *Eli: Bare-metal performance for i/o virtualization*, SIGPLAN Not. 47 (2012), no. 4, 411–422.

[58] Irfan Habib, *Virtualization with kvm*, Linux Journal 2008 (2008), no. 166, 8.

[59] Simon Hansman and Ray Hunt, *A taxonomy of network and computer attacks*, Computers & Security 24 (2005), no. 1, 31 – 43.

[60] Kristi G Hawes, *Security content automation protocol*, (2009).

[61] Yanxiang He, Wei Chen, Min Yang, and Wenling Peng, *Ontology based cooperative intrusion detection system*, Network and Parallel Computing (Hai Jin, Guang Gao, Zhiwei Xu, and Hao Chen, eds.), Lecture Notes in Computer Science, vol. 3222, Springer Berlin / Heidelberg, 2004, 10.1007/978-3-540-30141-7_59, pp. 419–426.

[62] Gernot Heiser and Ben Leslie, *The okl4 microvisor: Convergence point of microkernels and hypervisors*, Proceedings of the first ACM asia-pacific workshop on Workshop on systems, ACM, 2010, pp. 19–24.

[63] Vincent C Hu, David Ferraiolo, Rick Kuhn, Arthur R Friedman, Alan J Lang, Margaret M Cogdell, Adam Schnitzer, Kenneth Sandlin, Robert Miller, Karen Scarfone, et al., *Guide to attribute based access control (abac) definition and considerations (draft)*, NIST special publication 800 (2013), no. 162.

[64] Vincent C Hu, D Richard Kuhn, and David F Ferraiolo, *Attribute-based access control*, Computer 48 (2015), no. 2, 85–88.

[65] Galen C. Hunt and James R. Larus, *Singularity: Rethinking the software stack*.

[66] Intel, *Intel trusted execution technology: Measured launched environment developer's guide*, revision 009 ed., June.

[67] Intel Corporation, *Intel trusted execution technology — preliminary architecture specification*, Tech. Report Document Number: 31516803, Intel Corporation, 2006.

[68] ISO ISO and IEC Std, *Iso 27002: 2005*, Information Technology-Security Techniques-Code of Practice for Information Security Management. ISO (2005).

[69] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim, *Sgx-bomb: Locking down the processor via rowhammer attack*, Proceedings of the 2nd Workshop on System Software for Trusted Execution, ACM, 2017, p. 5.

[70] Yan Jia, Yulu Qi, Huaijun Shang, Rong Jiang, and Aiping Li, *A practical approach to constructing a knowledge graph for cybersecurity*, Engineering 4 (2018), no. 1, 53–60.

[71] Joanna Rutkowska, *Qubes OS Architecture*, `http://qubes-os.org/files/doc/arch-spec-0.3.pdf`, 2010.

[72] Patrick Kamongi, Srujan Kotikela, Krishna Kavi, Mahadevan Gomathisankaran, and Anoop Singhal, *Vulcan: Vulnerability assessment framework for cloud computing*, Software Security and Reliability (SERE), 2013 IEEE 7th International Conference on, IEEE, 2013, pp. 218–226.

[73] Kbeast, *Kbeast*, 2012, http://zerosecurity.org/2012/05/kbeast-rootkit-2012.

[74] Eric Keller, Jakub Szefer, Jennifer Rexford, and Ruby B. Lee, *Nohype: Virtualized cloud infrastructure without the virtualization*, Proceedings of the 37th Annual International Symposium on Computer Architecture (New York, NY, USA), ISCA '10, ACM, 2010, pp. 350–361.

[75] Won Kim, *Cloud computing adoption*, International Journal of Web and Grid Services 7 (2011), no. 3, 225–245.

[76] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood, *sel4: Formal verification of an os kernel*, Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (New York, NY, USA), SOSP '09, ACM, 2009, pp. 207–220.

[77] John C Knight, *Safety critical systems: challenges and directions*, Software Engineer-

ing, 2002. ICSE 2002. Proceedings of the 24rd International Conference on, IEEE, 2002, pp. 547–550.

[78] S. Kotikela, S. Nimgaonkar, and M. Gomathisankaran, *Virtualization based secure execution and testing framework*, Parallel and Distributed Computing and Systems (Dallas, TX), IASTED, ACTA Press, 2011.

[79] Srujan Kotikela, Krishna Kavi, and Mahadevan Gomathisankaran, *Vulnerability assessment in cloud computing*, Proceedings of the International Conference on Security and Management (SAM), The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2012, p. 1.

[80] Srujan Kotikela, Tawfiq Shah, Mahadevan Gomathisankaran, and Gelareh Tabani, *Radium: Race-free on-demand integrity measurement architecture*, (2015).

[81] Daniel Krech, *Rdflib*, 2012.

[82] Dirk Leinenbach and Thomas Santen, *Verifying the microsoft hyper-v hypervisor with vcc*, International Symposium on Formal Methods, Springer, 2009, pp. 806–809.

[83] Flavio Lombardi and Roberto Di Pietro, *Secure virtualization for cloud computing*, Journal of Network and Computer Applications 34 (2011), no. 4, 1113–1122.

[84] Kalle Lyytinen and Youngjin Yoo, *Ubiquitous computing*, Communications of the ACM 45 (2002), no. 12, 63–96.

[85] Neil MacDonald, *Market guide for cloud workload protection platforms*, Gartner Research (2018).

[86] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft, *Unikernels: Library operating systems for the cloud*, SIGPLAN Not. 48 (2013), no. 4, 461–472.

[87] Anil Madhavapeddy and David J Scott, *Unikernels: Rise of the virtual library operating system*, Queue 11 (2013), no. 11, 30.

[88] Lakshay Malhotra, Devyani Agarwal, and Arunima Jaiswal, *Virtualization in cloud computing*, J. Inform. Tech. Softw. Eng 4 (2014), no. 2.

[89] Peter Maydell, *Qemu lite*, `https://github.com/intel/qemu-lite`, 2016.

[90] Bill McCarty, *Selinux: Nsa's open source security enhanced linux*, O'Reilly Media, Inc., 2004.

[91] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil D. Gligor, and Adrian Perrig, *Trustvisor: Efficient TCB reduction and attestation*, IEEE Symposium on Security and Privacy, IEEE Computer Society, 2010, pp. 143–158.

[92] Jonathan M. McCune, Bryan Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki, *Flicker: an execution infrastructure for tcb minimization*, EuroSys (Joseph S. Sventek and Steven Hand, eds.), ACM, 2008, pp. 315–328.

[93] Peter Mell, Tiffany Bergeron, and David Henning, *Creating a patch and vulnerability management program*, NIST Special Publication 800 (2005), 40.

[94] Peter Mell and Tim Grance, *Effectively and securely using the cloud computing paradigm*, NIST, Information Technology Laboratory 2 (2009), no. 8, 304–311.

[95] Peter Mell, Tim Grance, et al., *The nist definition of cloud computing*, (2011).

[96] Pascal Meunier, *Technical article wiley handbook of science and technology for homeland security, classes of vulnerabilities and attacks (2010), 2010-05-10 [pascal meunier].*

[97] Jon Meyer and Troy Downing, *Java virtual machine*, O'Reilly & Associates, Inc., 1997.

[98] Karissa Miller and Mahmoud Pegah, *Virtualization: virtually at the desktop*, Proceedings of the 35th annual ACM SIGUCCS fall conference, ACM, 2007, pp. 255–260.

[99] Chris Mitchell, *Trusted computing*, vol. 6, Iet, 2005.

[100] Thomas Morris, *Trusted platform module*, Encyclopedia of cryptography and security, Springer, 2011, pp. 1332–1335.

[101] Derek Gordon Murray, Grzegorz Milos, and Steven Hand, *Improving xen security through disaggregation*, Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, ACM, 2008, pp. 151–160.

[102] Mihir Sudarshan Nanavati, *Breaking up is hard to do : security and functionality in a commodity hypervisor*, November 2011, http://hdl.handle.net/2429/35591.

[103] National Institute of Standards and Technology, *The nist definition of cloud computing*, nist special publication 800-145 ed., January 2011.

[104] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig, *Intel virtualization technology: Hardware support for efficient processor virtualization.*, Intel Technology Journal 10 (2006), no. 3.

[105] Satyajeet Nimgaonkar, Srujan Kotikela, and Mahadevan Gomathisankaran, *Ctrust: A framework for secure and trustworthy application execution in cloud computing*, Cyber Security (CyberSecurity), 2012 International Conference on, IEEE, 2012, pp. 24–31.

[106] Steven Noel, Deborah Bodeau, and Rosalie McQuaid, *Big-data graph knowledge bases for cyber resilience*, Proceedings of the NATO IST-153/RWS-21 Workshop on Cyber Resilience (2017), 6–21.

[107] Martin O'Connor, *Swrl tab*, 2012.

[108] oslo, *Oslo: Improving the security of trusted computing*, Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium (Berkeley, CA, USA), SS'07, USENIX Association, 2007, pp. 16:1–16:9.

[109] Claus Pahl, *Containerization and the paas cloud*, IEEE Cloud Computing 2 (2015), no. 3, 24–31.

[110] T Palmaers, *Implementing a vulnerability management process*, SANS Institute Reading Room (2013).

[111] Bryan Jeffrey Parno, *Trust extension as a mechanism for secure code execution on commodity computers*, (2010).

[112] Andrew Simmonds Peter, Peter S, and Louis Van Ekert, *An ontology for network security attacks*, In Proceedings of the 2nd Asian Applied Computing Conference (AACC04), LNCS 3285, Springer-Verlag, 2004, pp. 317–323.

[113] Nick L. Petroni, Jr. Timothy, Fraser Jesus, Molina William, and A. Arbaugh, *Copilot - a coprocessor-based kernel runtime integrity monitor*, In Proceedings of the 13th USENIX Security Symposium, 2004, pp. 179–194.

[114] Steve Ragan, *Adultfriendfinder data breach*, `https://www.csoonline.com/article/`

3139311/security/412-million-friendfinder-accounts-exposed-by-hackers.html, 2016.

[115] ———, *Equifax data breach*, https://www.csoonline.com/article/3223229/security/equifax-says-website-vulnerability-exposed-143-million-us-consumers.html, 2017.

[116] Reiner Sailer and Ronald Perez Stefan Berger and Ramn Cceres and Leendert van Doorn, *Towards Enterprise-level Security with Xen*, 2006.

[117] Mendel Rosenblum, *The reincarnation of virtual machines*, Queue 2 (2004), no. 5, 34.

[118] Mendel Rosenblum and Tal Garfinkel, *Virtual machine monitors: Current technology and future trends*, Computer 38 (2005), no. 5, 39–47.

[119] Daniel Rossier, *Embeddedxen: A revisited architecture of the xen hypervisor to support arm-based embedded virtualization*, White paper, Switzerland (2012).

[120] Margaret Rouse, *Security cia triad*, 2015.

[121] Jyotiprakash Sahoo, Subasish Mohapatra, and Radha Lath., *Virtualization: A survey on concepts,taxonomy and associated security issues.*, Second International Conference on Computer and Network Technology. (2010).

[122] Kristian Sandström, Aneta Vulgarakis, Markus Lindgren, and Thomas Nolte, *Virtualization technologies in embedded real-time systems*, Emerging Technologies & Factory Automation (ETFA), 2013 IEEE 18th Conference on, IEEE, 2013, pp. 1–8.

[123] Ariel Segall, *Using the tpm: Machine authentication and attestation*, (2012).

[124] Sean W Smith and Steve Weingart, *Building a high-performance, programmable secure coprocessor*, Computer Networks 31 (1999), no. 8, 831–860.

[125] Raghunathan Srinivasan, Prashant Dewan, Partha Dasgupta, Ravi Sahita, Uday Savagaonkar, and David Durham, *Mivmm: A micro vmm for development of a trusted code base.*

[126] Malgorzata Steinder, Ian Whalley, David Carrera, Ilona Gaweda, and David Chess, *Server virtualization in autonomic management of heterogeneous workloads*, Integrated

Network Management, 2007. IM'07. 10th IFIP/IEEE International Symposium on, IEEE, 2007, pp. 139–148.

[127] Geoffrey Strongin, *Trusted computing using AMD "pacifica" and "presidio" secure virtual machine technology*, Inf. Sec. Techn. Report 10 (2005), no. 2, 120–132.

[128] G.E. Suh, C.W. O'Donnell, and S. Devadas, *Aegis: A single-chip secure processor*, International Symposium on Information Science and Engineering (2008).

[129] Nancy Sumrall and Manny Novoa, *Trusted computing group (tcg) and the tpm 1.2 specification*, Intel Developer Forum, vol. 32, 2003.

[130] Takeshi Takahashi, Youki Kadobayashi, and Hiroyuki Fujiwara, *Ontological approach toward cybersecurity in cloud computing*, Proceedings of the 3rd international conference on Security of information and networks (New York, NY, USA), SIN '10, ACM, 2010, pp. 100–109.

[131] Hongwei Tang, Shengzhong Feng, Xiaofang Zhao, and Yan Jin, *Virtav: an agentless runtime antivirus system for virtual machines*, KSII Transactions on Internet and Information Systems (TIIS) 11 (2017), no. 11, 5642–5670.

[132] Trusted Computing Group, *TPM main specification*, Main Specification Version 1.2 rev. 85, Trusted Computing Group, February 2005.

[133] TrustedGRUB, Online, Last accessed in 03/2014 http://projects.sirrix.com/trac/trustedgrub.

[134] R. Uhlig, G. Neiger, D. Rodgers, A. Santoni, C. Fernando, A. Andrew, S. Bennett, A. Kagi, F. Leung, and L. Smith, *Intel virtualization technology*, Computer 38 (2005), 48–56.

[135] R. Uhlig, G. Neiger, D. Rodgers, A.L. Santoni, F.C.M. Martins, A.V. Anderson, S.M. Bennett, A. Kagi, F.H. Leung, and L. Smith, *Intel virtualization technology*, Computer 38 (2005), no. 5, 48 – 56.

[136] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L Santoni, Fernando CM Martins, Andrew V Anderson, Steven M Bennett, Alain Kagi, Felix H Leung, and Larry Smith, *Intel virtualization technology*, Computer 38 (2005), no. 5, 48–56.

[137] Jeffrey Undercoffer, Anupam Joshi, and John Pinkston, *Modeling computer attacks: An ontology for intrusion detection*, Recent Advances in Intrusion Detection (Giovanni Vigna, Christopher Kruegel, and Erland Jonsson, eds.), Lecture Notes in Computer Science, vol. 2820, Springer Berlin Heidelberg, 2003, pp. 113–135.

[138] Leendert Van Doorn, *Hardware virtualization trends*, ACM/Usenix International Conference On Virtual Execution Environments: Proceedings of the 2 nd international conference on Virtual execution environments, vol. 14, 2006, pp. 45–45.

[139] Amit Vasudevan, Sagar Chaki, Limin Jia, Jonathan M. McCune, James Newsome, and Anupam Datta, *Design, implementation and verification of an extensible and modular hypervisor framework*, IEEE Symposium on Security and Privacy, IEEE Computer Society, 2013, pp. 430–444.

[140] Amit Vasudevan, Bryan Parno, Ning Qu, Virgil D. Gligor, and Adrian Perrig, *Lockdown: Towards a safe and practical architecture for security applications on commodity platforms*, Proceedings of the 5th International Conference on Trust and Trustworthy Computing (Berlin, Heidelberg), TRUST'12, Springer-Verlag, 2012, pp. 34–54.

[141] Bill Venners, *The java virtual machine*, McGraw-Hill, New York, 1998.

[142] Volatility, *Volatility*, 2014, http://code.google.com/p/volatility/.

[143] W3C, *OWL Web Ontology Language* .

[144] Xin Wan, Zhiting Xiao, and Yi Ren, *Building trust into cloud computing using virtualization of tpm*, Multimedia Information Networking and Security (MINES), 2012 Fourth International Conference on, Nov 2012, pp. 59–63.

[145] Chonghua Wang, Zhiyu Hao, and Xiaochun Yun, *Nor: Towards non-intrusive, real-time and os-agnostic introspection for virtual machines in cloud environment*, International Conference on Information Security and Cryptology, Springer, 2017, pp. 500–517.

[146] Guohui Wang and TS Eugene Ng, *The impact of virtualization on network performance of amazon ec2 data center*, Infocom, 2010 proceedings ieee, IEEE, 2010, pp. 1–9.

[147] Ju An Wang and Minzhe Guo, *Ovm: an ontology for vulnerability management*, Pro-

ceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies (New York, NY, USA), CSIIRW '09, ACM, 2009, pp. 34:1–34:4.

[148] Ju An Wang, Minzhe Guo, Hao Wang, Min Xia, and Linfeng Zhou, *Ontology-based security assessment for software products*, Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies (New York, NY, USA), CSIIRW '09, ACM, 2009, pp. 15:1–15:4.

[149] Wei Wang, Rong Jiang, Yan Jia, Aiping Li, and Yi Chen, *Kgbiac: Knowledge graph based intelligent alert correlation framework*, International Symposium on Cyberspace Safety and Security, Springer, 2017, pp. 523–530.

[150] Jon Watson, *Virtualbox: bits and bytes masquerading as machines*, Linux Journal 2008 (2008), no. 166, 1.

[151] Timothy Grance Wayne Jansen, *Guidelines on security and privacy in public cloud computing*, Jan 2011.

[152] Steve R. White and Liam Comerford, *Abyss: An architecture for software protection.*, IEEE Transactions on Software Engineering 16 (1990), no. 6.

[153] Steve R White, Steve H Weingart, William C Arnold, and Elaine R Palmer, *Introduction to the citadel architecture: Security in physically exposed environments*, Tech. report, Technical Report RC16672, Distributed security systems group, IBM Thomas J. Watson Research Center, 1991.

[154] Monty Wiseman, *Trusted computing group*, (2004).

[155] www.wikipedia.org, *Ontology (Information Science)*.

[156] L. Xiaoqi and S. Smith, *A microkernel virtual machine:: building security with clear interfaces*, Proceedings of the 2006 workshop on Programming languages and analysis for security (New York, NY, USA), PLAS '06, ACM, 2006, pp. 47–56.

[157] Xiongwei Xie and Weichao Wang, *Rootkit detection on virtual machines through deep*

information extraction at hypervisor-level, Communications and Network Security (CNS), 2013 IEEE Conference on, Oct 2013, pp. 498–503.

[158] Yuping Xing and Yongzhao Zhan, *Virtualization and cloud computing*, Future Wireless Networks and Information Systems, Springer, 2012, pp. 305–312.

[159] Jun Yang, Youtao Zhang, and Lan Gao., *Fast secure processor for inhibiting software piracy and tampering*, Proceedings of the 36th International Symposium on Microarchitecture (MICRO-3603) (2003).

[160] Bennet Yee, *Using secure coprocessors*, Tech. report, CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, 1994.

[161] Andrew J. Younge, Robert Henschel, James T. Brown, Gregor von Laszewski, Judy Qiu, and Geoffrey C. Fox, *Analysis of virtualization technologies for high performance computing environments*, Cloud Computing (CLOUD), 2011 IEEE International Conference on, july 2011, pp. 9 –16.

[162] Xiaotong Zhuang, Tao Zhang, and Santosh Pande, *Hide: An infrastructure for efficiently protecting information leakage on the address bus*, ASPLOS (2004).

[163] Vincent J. Zimmer, Shiva R. Dasari, and Sean P. Brogan, *Trusted platforms: Uefi, pi and tcg-based firmware*, Intel Corporation and IBM Corporation, September 2009, http://www.intel.com/content/dam/doc/white-paper/uefi-pi-tcg-firmware-white-paper.pdf.